

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

СЕРВЕРНІ ВЕБТЕХНОЛОГІЇ КОНСПЕКТ ЛЕКЦІЙ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за спеціальністю G7 Автоматизація, комп'ютерно-інтегровані
технології та робототехніка

Укладач: О. С. Бунке

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2026

УДК 004.738.5

Б91

Укладач: Бунке Олександр Сергійович, к.т.н., доцент

Рецензент Варава І.А., к.т.н., доцент кафедри інженерії програмного забезпечення в енергетиці КПІ ім. Ігоря Сікорського

Відповідальний
редактор Новіков П.В., к.т.н., доцент

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 1 від 11.09.2026 р.)
за поданням вченої ради ННІАТЕ
(протокол № 4 від 25.05.2026 р.)*

Б91

Серверні вебтехнології : конспект лекцій [Електронний ресурс] : навч. посіб. для студ. спеціальності G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка»; уклад.: О. С. Бунке. – Електронні текстові дані (1 файл: 4,8 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2026. – 143 с.

Посібник розроблений на підставі силабусу навчальної дисципліни «Серверні веб-технології» та призначений для проведення комп'ютерних практикумів, підвищення розуміння основ серверної веб-розробки.

Призначений для студентів, які навчаються за освітньою програмою підготовки бакалаврів за спеціальністю G7 "Автоматизація, комп'ютерно-інтегровані технології та робототехніка".

Спрямований на формування у студентів умінь та навичок з проєктування та розробки серверної частини веб-застосунків. Забезпечує студентів необхідними теоретичними знаннями для виконання практичних завдань, запланованих впродовж семестру.

УДК 004.738.5

Реєстр. № НП 26/27-003. Обсяг 5,4 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
<https://kpi.ua>

Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© КПІ ім. Ігоря Сікорського, 2026

Зміст

ВСТУП	6
Лекція 1. КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА	8
1.1 Вступ	8
1.2 Протокол HTTP та опис його роботи.....	11
1.3 Протокол HTTPs.....	12
1.4 Бекенд: огляд та інструменти	12
1.5 Application Programming Interface (API)	14
1.6 REST API: архітектура та робота	16
1.7 Формати обміну даними: XML та JSON.....	18
1.8 Файли-куки	19
1.9 Сесії	20
1.10 Автентифікація: сесії vs токени.....	21
1.11 Питання та завдання для самоконтролю.....	22
Лекція 2. ТИПИ ВЕБ-САЙТІВ	23
2.1 Статичні сайти на зорі Інтернету.....	23
2.2 Технологія AJAX.....	24
2.3 Single Page Application.....	24
2.4 Повернення до Server Side Rendering.....	25
2.5 Static Site Generator	27
2.6 Питання та завдання для самоконтролю.....	29
Лекція 3. ОГЛЯД МОВ ДЛЯ СЕРВЕРНИХ ЗАСТОСУНКІВ	30
3.1 Типи мов програмування.....	30
3.2 HTML та CSS.....	31
3.3 Мова програмування Python	32
3.4 Мова програмування JavaScript.....	32
3.5 Мова програмування PHP	33
3.6 Мова програмування Java	34
3.7 Мова програмування C#.....	35
3.8 Питання та завдання для самоконтролю.....	35
Лекція 4. СЕРВІСНА АРХІТЕКТУРА	36
4.1 Основні принципи мікросервісної архітектури	36
4.2 Переваги мікросервісної архітектури	37
4.3 Монолітна архітектура	37
4.4 Порівняння сервісного та монолітного підходів	39
4.5 Перехід від моноліту до мікросервісів.....	39
4.6 Контейнеризація як інструмент сервісної архітектури	40
4.7 Брокери повідомлень: асинхронна комунікація між сервісами	40
4.8 Підсумки до лекції	42

	4
4.9 Питання та завдання для самоконтролю.....	43
Лекція 5. ОСНОВИ МОВИ PHP.....	44
5.1 Вступ	44
5.2 Основи мови PHP: змінні, константи, функції.....	45
5.3 Робота з даними: посилання, масиви, генератори	47
5.4 Чистий код: найменування, функції та форматування.....	49
5.5 Чистий код: класи, помилки та межі.....	52
5.6 Приклади коду PHP	53
5.7 Сучасний синтаксис PHP 8.x	57
5.8 Об'єктно-орієнтоване програмування у PHP.....	61
5.9 Практичні веб-прикладі.....	66
5.10 Стандарти PSR.....	68
5.11 Питання та завдання для самоконтролю.....	71
Лекція 6. ПАКЕТНИЙ МЕНЕДЖЕР COMPOSER.....	73
6.1 Вступ	73
6.2 Інсталяція Composer.....	73
6.3 Встановлення на Linux/Unix/macOS	74
6.4 Встановлення на Windows.....	74
6.5 Синтаксис та опції Composer	75
6.6 Структура composer.json та composer.lock.....	76
6.7 Семантичне позначення версій.....	77
6.8 Оновлення пакетів.....	78
6.9 Видалення пакетів.....	78
6.10 Скидання автозавантаження	78
6.11 Питання та завдання для самоконтролю.....	78
Лекція 7. ФРЕЙМВОРК SYMFONY	80
7.1 Основні компоненти фреймворка Symfony.....	80
7.2 Структура веб-додатку на Symfony.....	81
7.3 Процес запиту.....	82
7.4 Розгортання проєкту на Symfony	82
7.5 Контролери та маршрутизація	83
7.6 Doctrine ORM.....	86
7.7 Сервіси та Dependency Injection.....	89
7.8 Валідація та серіалізація.....	91
7.9 Тестування у Symfony.....	94
7.10 Питання та завдання для самоконтролю.....	95
Лекція 8. СЕРВЕРНИЙ JAVASCRIPT: NODE.JS.....	97
8.1 Вступ	97
8.2 Блокуючі введення/виведення та проблема C10K.....	98
8.3 Node.js: платформа та цикл подій.....	100

8.4	CPU-ємні завдання та воркери.....	104
8.5	Директиви Import та Export.....	109
8.6	Cookie та сесії в Node.js.....	110
8.7	npm та package.json	111
8.8	TypeScript: основи для серверної розробки.....	113
8.9	NestJS та Prisma ORM.....	114
8.10	Створення проєкту та структура NestJS	116
8.11	REST API у NestJS: контролери та сервіси.....	118
8.12	Робота з базою даних: Prisma ORM.....	120
8.13	Валідація та DTO у NestJS	123
8.14	Гварди, перехоплювачі та JWT-аутентифікація	125
8.15	Питання та завдання для самоконтролю до Розділу 8.....	128
Лекція 9. РОЗГОРТАННЯ ВЕБ-ДОДАТКІВ У МЕРЕЖІ		130
9.1	Вступ	130
9.2	Розгортання простих PHP-сайтів через FTP.....	130
9.3	Контейнеризація з Docker	131
9.4	GitLab CI	137
9.5	GitHub Actions	138
9.6	Питання та завдання для самоконтролю.....	140
ПЕРЕЛІК ПОСИЛАНЬ		142
ПЕРЕЛІК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ.....		143

ВСТУП

Практично всі веб-сайти та мобільні застосунки звідкись отримують і кудись зберігають дані – ця непомітна для користувача компонента називається серверною, або «бек-ендом». Розробка серверних продуктів для вебу базується на знаннях архітектури, баз даних та деяких популярних мов програмування.

У посібнику наводиться опис принципів роботи серверних веб-застосунків, порядок взаємодії користувача із веб-сервером, протокол HTTP, принципи проєктування програмних інтерфейсів (REST API, OpenAPI), архітектурні стилі веб-застосунків, реляційні бази даних, сучасний синтаксис PHP 8.x та фреймворк Symfony 7, серверний JavaScript на Node.js 20 LTS і фреймворк NestJS 10 з Prisma ORM, контейнеризація засобами Docker та CI/CD через GitHub Actions.

Метою навчальної дисципліни «Серверні веб-технології» є формування та розвиток у студентів таких компетентностей:

- здатність застосовувати знання у практичних ситуаціях;
- здатність до пошуку, опрацювання та аналізу інформації з різних джерел;
- здатність використовувати для вирішення професійних завдань новітні технології у галузі автоматизації та комп'ютерно-інтегрованих технологій, зокрема, проєктування багаторівневих систем керування, збору даних та їх архівування для формування бази даних параметрів процесу та їх візуалізації за допомогою засобів людино-машинного інтерфейсу.

- здатність вільно користуватись сучасними комп'ютерними та інформаційними технологіями для вирішення професійних завдань, програмувати та використовувати прикладні та спеціалізовані комп'ютерно-інтегровані середовища для вирішення задач автоматизації.

У результаті вивчення навчальної дисципліни студенти набудуть таких загальних програмних результатів навчання:

- вміти програмувати інформаційні сервіси з використанням клієнт-серверних технологій.

Для успішного опанування дисципліни студент має володіти базовими знаннями з алгоритмів, прослухати дисципліни «сучасні технології програмування» та «проєктування та розробка баз даних».

Результати навчання використовуються при дипломному проєктуванні, у вибіркових дисциплінах подібного спрямування та у професійній діяльності.

Технологічний стек курсу. Навчальний посібник редакції 2026 року спирається на сучасні стабільні версії галузевих інструментів: PHP 8.3, Symfony 7.x з Doctrine ORM, Node.js 20 LTS, NestJS 10+ з Prisma 5 ORM та TypeScript 5, MySQL 8, Docker 24 з Docker Compose V2, а також GitHub Actions як основну платформу неперервної інтеграції. Обрані версії відображають поточну ситуацію в індустрії та вимоги до випускників, актуальні на 2026 рік.

Лекція 1. КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА

План лекції:

1. Вступ
2. Протокол HTTP та опис його роботи
3. Протокол HTTPS
4. Бекенд: огляд та інструменти
5. Application Programming Interface (API)
6. REST API: архітектура та робота
7. Формати обміну даними: XML та JSON
8. Файли-куки
9. Сесії
10. Автентифікація: сесії vs токени

Ключові слова: HTTP, API, обмін даними, бекенд, фронтенд.

1.1 Вступ

Функціонування мережі Інтернет засноване на архітектурі клієнт-сервер. Водночас програма-клієнт (зазвичай браузер) на комп'ютері користувача надсилає запити для отримання інформації або ресурсів (зазвичай це запити за URL-адресою) програмі-серверу, що встановлена на одному із серверних комп'ютерів, підключених до мережі та ідентифікованому IP-адресою. Логіка роботи сервера забезпечує видачу файлів, контенту, ресурсів, які зберігаються на сервері, браузеру і кілька допоміжних функцій (рис. 1.1) [1].

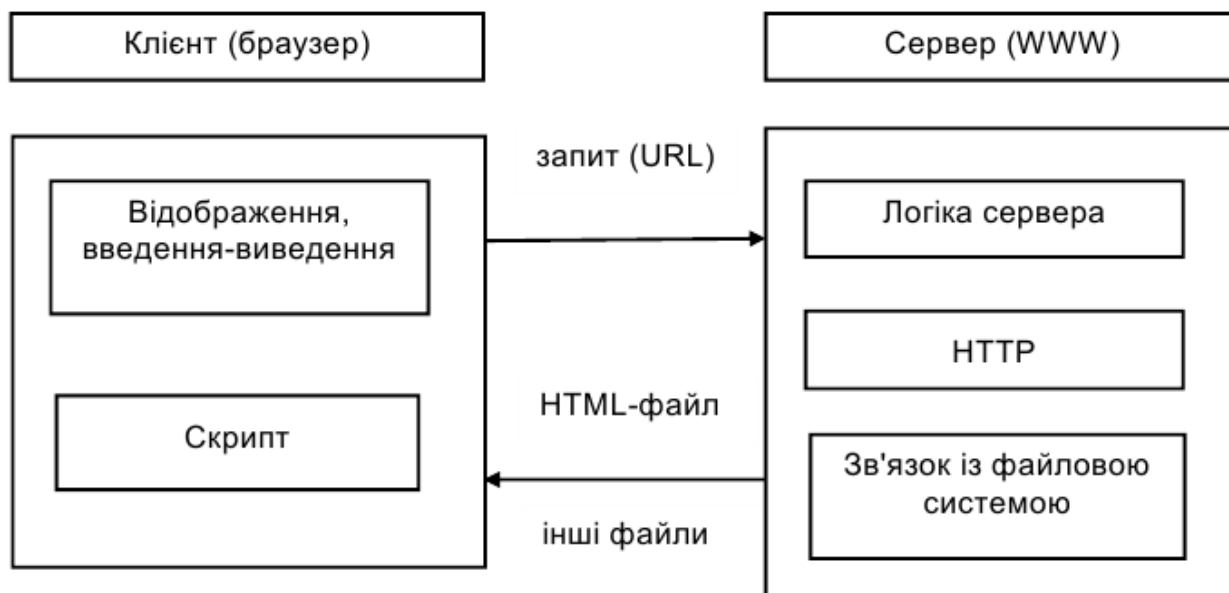


Рисунок 1.1 – Клієнт-серверна архітектура у WEB

За такої взаємодії сервер грає роль сховища. Для розширення можливостей веб-сторінок із реалізації більш складних функцій необхідно розширювати логіку сервера за рахунок виконання спеціально розроблених програм. Ці програми можуть виконувати найрізноманітніші функції: оброблення сторінок перед відправкою, зв'язок із базами даних, проведення обчислень тощо. Таким чином під серверними технологіями будемо розуміти сукупність засобів і методів запуску виконуваних програм на сервері.

Для забезпечення функціонування сервісу WWW було розроблено досить багато серверів. Вони володіють різними характеристиками, але найбільш масовими є два: IIS (Internet Information Services), розроблений компанією "Майкрософт", і вільно поширювані Apache та Nginx.

Програми, які розробляються для виконання на сервері крім реалізації основних функцій, повинні мати кілька особливостей (порівняно з тими, що виконуються в середовищі операційної системи):

- запуск здійснюється сервером;
- має бути забезпечена можливість взаємодії із сервером;
- має бути забезпечена можливість надсилати результати (дані) клієнту, який видав запит (браузеру).

Для створення серверних розширень було розроблено кілька інтерфейсів (специфікацій на взаємодію між сервером та програмою), найбільшого поширення набули: CGI, FCGI і API-інтерфейси. Кожен із них має свої переваги, так само, як і певні недоліки [1].

Технологія активних серверних сторінок (включення коду в саму HTML-сторінку) полягає в тому, що запитувана сторінка з розміченим текстом містить програмний код, який виконується на сервері перед надсиланням клієнту, змінюючи запитувану сторінку. Цей код може отримувати дані з бази даних, будувати користувацький інтерфейс залежно від отриманих даних, а потім надсилати HTML-сторінку назад клієнту. Кінцевим результатом такого процесу буде надсилання браузеру стандартної веб-сторінки на HTML. Цей процес є динамічне створення веб-сторінок. Сторінка, яка показується в браузері, насправді є продуктом виконання коду, який був запущений після клієнтського запиту, на сервері такої сторінки фізично не існує.

Гнучкість і зручність веб-додатків ґрунтується на відокремленні візуальної частини застосунку від логіки його роботи. веб-додаток по суті є розподіленою програмою, яка використовує для обміну даними протокол HTTP. Перевагою такого додатка є стандартний, всім відомий клієнт-браузер, і зниження витрат на супровід додатків розробниками. Замість того, щоб встановлювати додаток Windows на кожне робоче місце в компанії, можна створити один веб-додаток, до якого є загальний доступ. Якщо виникає необхідність внесення в цей додаток змін або виправлень, то досить оновити додаток тільки на сервері, а не на всіх клієнтських комп'ютерах. Це також дозволяє кожному клієнту постійно користуватися самою останньою версією програми, оскільки саме вона завжди буде знаходитися на сервері [1].

Незважаючи на те, що мова йде, по суті, про розподілений додаток, користувач, який працює в браузері (набираючи URL в рядку адреси або клацаючи по посиланнях), може цього не помічати. У зв'язку із цим додатки, що складаються з активних сторінок, продовжують називати сайтами.

1.2 Протокол HTTP та опис його роботи

HTTP — це протокол передачі даних, на основі якого працює всесвітня павутина. Завдяки цьому протоколу ми можемо заходити на сайти в браузері та взаємодіяти з ними: переходити з однієї сторінки на іншу, завантажувати файли та переглядати зображення, обмінюватися повідомленнями та оплачувати покупки.

Абревіатура HTTP розшифровується як HyperText Transfer Protocol — протокол передавання гіпертексту (тобто текстових документів, які містять посилання інші документи). Нині за його допомогою передають будь-які формати даних.

Протокол HTTP використовує в роботі технологію клієнт-сервер: клієнт надсилає на сервер запит, де спеціальна програма його обробляє, формує відповідь і повертає клієнту.

У ролі клієнта зазвичай виступає браузер, але його функції може виконувати й інша програма. Наприклад, пошуковий робот, який подорожує сайтами і сканує їх для індексації в пошукових системах. У ролі сервера виступає веб-сервер — спеціальна програма на фізичному сервері, де зберігається сайт.

Наприклад, щоб відобразити будь-яку сторінку, ваш браузер надсилає на сервер запит на отримання HTML-документа з її вмістом. Далі браузер вивчає документ і запитує додаткові файли, необхідні для відображення сторінки: стилі елементів, зображення, скрипти. Потім браузер збирає все це разом, як зазначено в HTML-документі, та виводить на екран комп'ютера результат [2].

Інший приклад — коли браузер надсилає якісь дані серверу. Скажімо, логін та пароль для входу до облікового запису. Щоб сформувати HTML-документ з головною сторінкою облікового запису, сервер перевіряє деталі входу в спеціальній таблиці всередині бази даних, де зберігаються дані всіх користувачів. Якщо у таблиці є користувач із зазначеними даними, сервер формує HTML-документ і надсилає його браузеру, як у прикладі вище. Якщо такого користувача немає або деталі входу неправильні, у рушії сайту прописаний спеціальний сценарій на цей випадок — надіслати повідомлення з помилкою, яку браузер відобразить на екрані.

1.3 Протокол HTTPS

HTTPS — це розширення для протоколу HTTP, що робить його безпечним. Річ у тім, що дані передаються за HTTP у відкритому вигляді. Це створює ризик розкрити конфіденційну інформацію, якщо хтось перехопить трафік. HTTPS вирішує цю проблему, додаючи в початковий протокол можливість шифрування даних [3].

Абревіатура HTTPS розшифровується як HyperText Transfer Protocol Secure — безпечний протокол передавання гіпертексту. Безпека досягається за рахунок об'єднання протоколу HTTP з криптографічним протоколом TLS [2] (рис. 1.2).

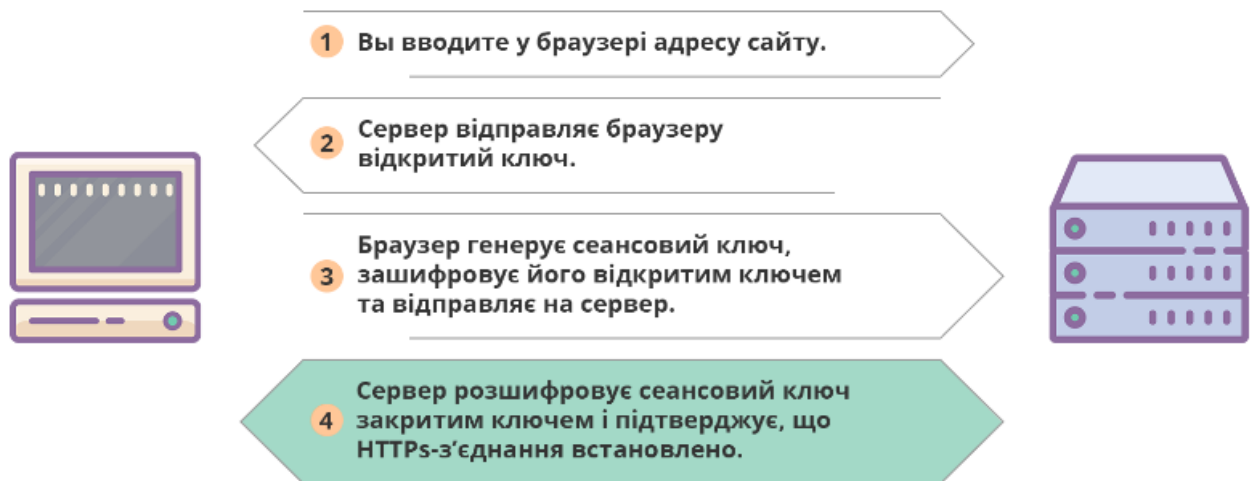


Рисунок 1.2 – Обмін запитами по HTTPS

1.4 Бекенд: огляд та інструменти

Бекенд — це узагальнене поняття програмно-апаратного комплексу серверної частини веб-застосунку. Простіше кажучи, це все, що працює на сервері. Виходячи з цього, бекенд-розробка — це робота над програмними засобами, спрямованими на реалізацію логіки ресурсу [1]. Ця частина прихована від очей користувача, оскільки відбувається за межами його браузера або конкретно взятого комп'ютера. Розробник сайтів у цьому випадку використовує ті ресурси, які є на сервері. При цьому його обов'язки можуть значно варіюватися, залежно від того про який продукт йде мова. Так, фахівець може займатися створенням,

інтеграцією баз даних, забезпечувати безпеку ресурсу, налаштовувати технології резервного копіювання або ж відновлення інформації [3].

Для створення серверної частини сайту необхідно освоїти повноцінну мову програмування. Вона може бути практично будь-якою, але зараз у веб-розробці найчастіше використовують PHP, JavaScript та Python [3]. Це мови загального призначення, але для створення сайтів вони підходять у більшості випадків. Для розробки серверної частини необхідно розібратися з базами даних. Найпопулярніші СУБД для розробки веб-додатків: MySQL, PostgreSQL, MongoDB, також для хмарних швидких рішень використовується платформа Google Firebase [4].

Сучасна розробка бекенду - все частіше це розробка програмного інтерфейсу (API), яким користуються як браузерери так і мобільні додатки [3].

Для розробки повноцінного серверного додатку зазвичай використовуються підготовлені інструментарії з набором вже стабільних протестованих компонентів та структурою проєкту - фреймворки.

Для бекенду рекомендується застосовувати:

- PHP 8.3: Symfony 7.x (основний у курсі), Laravel 11
- Node.js 20 LTS: NestJS 10+ (основний у курсі), Express
- Python: Django 5, Flask (довідково)

Основні відмінності цих понять полягають у тому, що одне з них (фронтенд) охоплює все, що бачить користувач при роботі з сайтом, а інше (бекенд), навпаки, перебуває поза полем зору людини. Це дві частини того самого проєкту, одного цілого, і є кілька варіантів, як вони будуть взаємодіяти між собою [4].

Зазвичай весь процес проходить циклічно:

- Frontend збирає призначені для користувача дані і перенаправляє їх в Backend;
- відбувається обробка даних;
- інформація повертається, прийнявши зрозумілу форму і виконавши запит.

Відмінності Frontend від Backend сайту істотні, так як за кожно з названих вище задачу відповідає окремий фахівець, а успішний результат можливий тільки при взаємодоповнюючій командній роботі.

Зрозуміти особливості їх взаємодії найпростіше на прикладі. Так, оплачуючи покупку в інтернеті, ви заповнюєте дані своєї карти, натискаєте кнопку «оплатити» і отримуєте сповіщення про те, що оплата пройшла. Це чистої води фронтенд. А ось як далі гроші рухаються по мережі, як продавець отримує ваше замовлення – ви не бачите, це бекенд.

1.5 Application Programming Interface (API)

Абревіатура API розшифровується як Application Programming Interface або програмний інтерфейс застосунку. API – це набір способів і правил взаємодії та обміну даними між різними програмами [3]. Їх спілкування, якщо можна так висловитися, відбувається за допомогою класів, методів, функцій і структур. Іноді комунікації можливі за допомогою констант однієї програми, до яких звертаються інші. Говорячи простими словами, API виступає в ролі посередника, що дозволяє програмі 1 отримати доступ до даних і навіть деяким можливостям програми 2. Все це дозволяє розробникам поліпшити функціональність продуктів, що випускаються і пов'язати їх з іншими додатками.

Щоразу, коли користувач відвідує будь-яку сторінку в мережі, він взаємодіє з API віддаленого сервера. API-це складова частина сервера, яка отримує запити і надсилає відповіді.

Інтерфейс – це межа між двома програмами, за допомогою якої вони можуть обмінюватися інформацією і виконувати пов'язані один з одним функції. При цьому процеси, що відбуваються всередині кожної з них, недоступні для іншої програми. Схоже визначення раніше ми дали для API. Завдяки такому підходу вдається налагодити взаємодію між декількома програмами, не переймаючись питанням про їх устрій, програмну логіку або способи обробки даних [2] (див. ілюстрацію на рис. 1.3). За допомогою інтерфейсів програмістам не потрібно розбиратися в програмному коді інших фахівців, щоб підключити

свій продукт до іншого. Користувачам же при цьому не потрібно замислюватися про те, що стоїть за звичними функціями в девайсах. Наприклад, для відправки повідомлення зі смартфона їм не потрібно знати принцип роботи тачскріна. Замість цього достатньо набрати потрібний текст, натиснувши певне поєднання віртуальних клавіш. API дозволяє користуватися функціями програми, не маючи уявлення про те, як вона працює. З цієї причини даний термін прирівнюють до інтерфейсу.

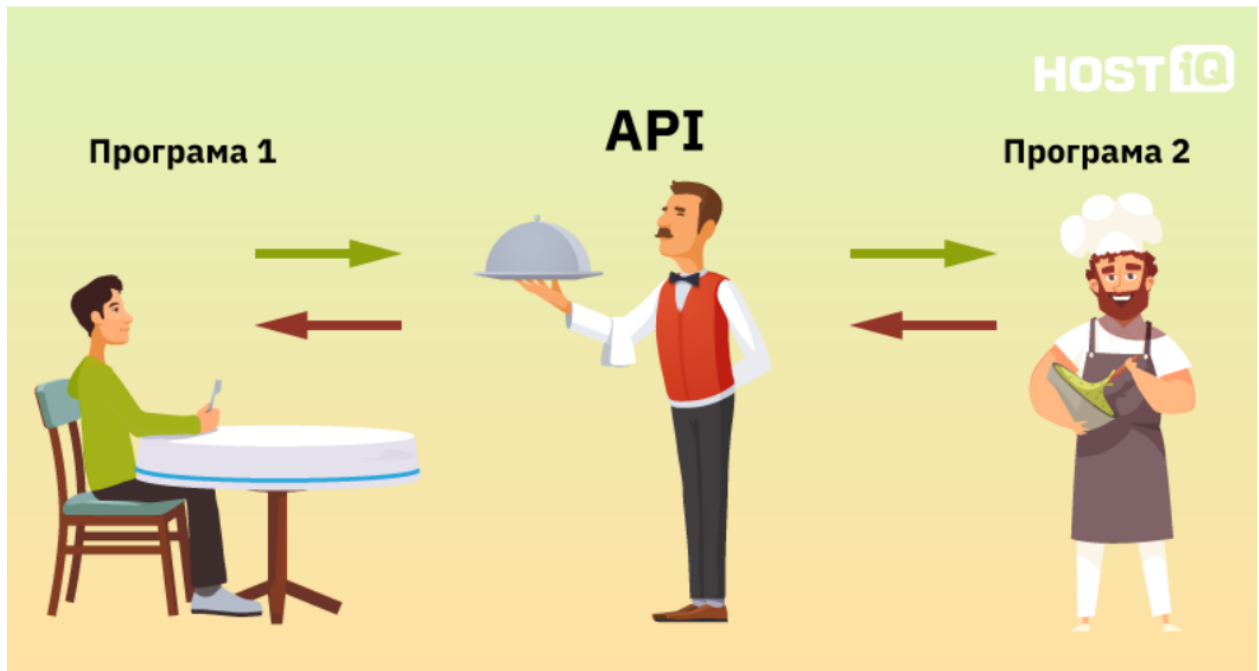


Рисунок 1.3 – Суть роботи API на прикладі обслуговування в ресторані

Головне завдання API полягає в тому, щоб налагодити взаємодію між певними утилітами і не замислюватися при цьому, як вони влаштовані або як вони обробляють дані.

Але у подібного інструменту є і інші цілі:

1. зробити програмне забезпечення більш безпечним;
2. заробити на тому, що інші розробники використовують функціонал готового продукту для удосконалення власних напрацювань.

Інтерес програмістів до API обумовлений наступними причинами:

1. збільшення безпеки розробки. За допомогою API можна винести в окремий додаток той функціонал, який повинен бути захищений від стороннього втручання. Це знижує ймовірність некоректного його використання сторонніми програмами;

2. зниження вартості розробки і економія часу. Купити готовий API іноді вигідніше, ніж витратити час на створення функціоналу з нуля з усіма наслідками, що випливають звідси наслідками. Можна користуватися вже готовими інструментами, а не винаходити велосипед з нуля;
3. спрощення зв'язків між різними системами і сервісами. Розробники можуть впроваджувати у свої продукти підтримку сторонніх сервісів, навіть не замислюючись про те, хто їх створював.

Наприклад, розробник розробив сайт із онлайн-конвертером валют. Він хоче, щоб конвертером користувалися не лише на вихідному сайті, а й у банківських програмах.

Як це зробити:

1. розробник конвертера створює API та викладає його на своєму сайті. Описує в документації, як працює API та які в нього є функції;
2. розробники банківських програм беруть API конвертера та додають його до свого коду, якщо хочуть дати своїм користувачам функції цього конвертера;
3. користувач програми вводить суму, яку хоче «обміняти», та вибирає валюту;
4. програма надсилає на сервер конвертера спеціальний запит із сумою, яку потрібно перевести у вибрану валюту;
5. конвертер перераховує суму та надсилає відповідь з результатом у потрібній валюті;
6. програма отримує відповідь та показує її користувачу.

У результаті всі задоволені: користувач переклав цифри в іншу валюту в додатку, конвертером користуються не лише на сайті, а банківські програми не писали конвертер з нуля, але покращили свій продукт.

1.6 REST API: архітектура та робота

REST або **Representational state transfer**, що перекладається як Передача Репрезентативного Стану — це архітектурний стиль проєктування API з використанням протоколу HTTP [5]. Давайте розглянемо кожне слово окремо:

Репрезентативний (Representational): Тут мається на увазі, що дані (або "ресурси") можуть бути представлені в різних форматах, таких як JSON, XML тощо.

Стан (State): Вказує на стан ресурсу на сервері, який може бути змінений або отриманий через HTTP-методи (GET, POST, PUT, DELETE, і т.д.).

Передача (Transfer): У цьому контексті означає взаємодію між клієнтом і сервером через HTTP-запити та відповіді.

Отже, "Передача Репрезентативного Стану" — це архітектурний підхід, який дозволяє взаємодіяти з ресурсами на сервері через стандартні HTTP-методи, і ці ресурси можуть бути представлені в різних форматах.

REST API застосовується скрізь, де є необхідність надання даних із сервера користувачеві веб-програми або сайту.

Головними компонентами REST API є:

Client – клієнт або програма, яка запущена на стороні користувача (на його девайсі) та ініціює комунікацію.

Server — сервер, який надає API як доступ до своїх даних та функцій.

Resource - ресурс є будь-яким видом контенту (відео, текст, картинка), який сервер може передати клієнту.

REST API взаємодіє з допомогою HTTP запитів, виконуючи стандартні функції: створення, оновлення, читання, видалення записів у ресурсі. Існує 6 методів, що описують, що потрібно робити з ресурсом [1]:

GET: Використовується для отримання даних від сервера. Не змінює стан ресурсу.

Приклад: Отримати список користувачів.

POST: Використовується для створення нового ресурсу на сервері. Дані для створення ресурсу передаються в тілі запиту.

Приклад: Додати нового користувача.

PUT: Використовується для оновлення існуючого ресурсу. Дані для оновлення передаються в тілі запиту.

Приклад: Оновити інформацію про користувача.

PATCH: Схожий на PUT, але змінює лише ті поля ресурсу, які були вказані в запиті.

Приклад: Оновити електронну адресу користувача.

DELETE: Використовується для видалення ресурсу.

Приклад: Видалити користувача.

HEAD: Схожий на GET, але повертає лише заголовки відповіді, без тіла.

Приклад: Перевірити наявність ресурсу.

OPTIONS: Використовується для отримання інформації про методи, доступні для ресурсу.

Приклад: Дізнатися, які методи підтримуються для ресурсу.

Кожен з цих методів може повертати різні HTTP-статуси, які інформують клієнта про результат виконання запиту (наприклад, 200 OK, 201 Created, 400 Bad Request, 404 Not Found тощо).

RESTful API часто використовується для розробки веб-сервісів, які можуть бути легко інтегровані в різні системи, незалежно від мови програмування, платформи або типу пристрою.

1.7 Формати обміну даними: XML та JSON

Клієнт і сервер обмінюються даними у форматі, зрозумілому обом. Історично було два домінуючі формати для веб-API: XML та JSON. На 2026 рік JSON є де-факто стандартом для REST API в подавляючій більшості проєктів, а XML зберігається в легасі-інтеграціях та протоколах типу SOAP, RSS, Atom.

JSON (JavaScript Object Notation) — легковаговий формат обміну даними, який виник як підмножина синтаксису JavaScript, але став мовно-незалежним стандартом. Основні типи даних JSON: object, array, string, number, boolean, null. Приклад відповіді від REST-ендпойнту GET /api/products/42:

```
{ "id": 42, "title": "Ноутбук", "price": 24990.50, "in_stock": true, "tags":
["electronics", "laptop"] }
```

XML (eXtensible Markup Language) — розширювана мова розмітки, в якій дані оформлюються у вигляді вкладених тегів, аналогічних HTML. XML підтримує опис схем даних через XSD, простори імен, коментарі та атрибути.

Проте обсяг передаваних даних у 1,5–2 рази більший за JSON через дублювання імен відкриваючих та закриваючих тегів. Той самий приклад у XML:

```
<product id="42">
  <title>Ноутбук</title>
  <price>24990.50</price>
  <in_stock>true</in_stock>
  <tags>
    <tag>electronics</tag><tag>laptop</tag>
  </tags>
</product>
```

При формуванні HTTP-відповіді сервер вказує тип даних у заголовку Content-Type: application/json або application/xml. Слід розрізняти формати від протоколів: SOAP — це протокол на основі XML з жорсткою структурою (envelope/header/body), він вважається застарілим і практично повністю витіснений REST-підходом з JSON. У комп'ютерному практикумі № 3 передбачається робота виключно з JSON.

1.8 Файли-куки

Файли-куки (з англійської cookie перекладається як печиво) — це невеликі текстові файли, в яких зберігається інформація про наші дії на певних ресурсах [5].

Файли cookie корисні і для самого веб-ресурсу, і для користувача, і можуть виконувати такі функції:

- збір даних про переваги власника пристрою. Cookie дозволяють запропонувати йому тільки рекламу або сторінки ресурсу, які будуть затребувані.
- збереження налаштувань користувача та даних для входу на сайт. Це позбавляє нас необхідності щоразу заново вводити цю інформацію.
- прискорення серфінгу на сторінках веб-ресурсу.
- збір даних про затребуваність запропонованого продукту — рекламного чи інформаційного.

Механізм дії cookie дуже простий. Будь-яка наша дія на сайті фіксується та перенаправляється браузеру у вигляді куки, завдяки чому ресурс нас ідентифікує. Після того, як ми залишаємо сторінку, файли надсилаються назад.

Яку інформацію передають куки-файли.

Веб-ресурси за допомогою cookies збирають дані, які допоможуть їм краще просувати свої продукти. Це можуть бути [5]:

- технічні параметри, які вибирає користувач, – масштаб, шрифт, валюта, мова;
- товарні переваги – дані про переглянуту рекламу чи куплену продукцію;
- особисті дані, розташування та IP-пристрої для ідентифікації користувача;
- лайки та коментарі;
- тип ОС.

Усі ці файли аналізуються та роблять пошук інформації більш комфортним. Тому, якщо цей сайт використовує cookies, це не означає, що він намагається дізнатися особисті дані та нашкодити вам.

1.9 Сесії

Особливість веб-серверу полягає в тому, що він не здатний розпізнати, чи надходять запити від одного і того ж браузера, чи від різних, оскільки HTTP протокол не дозволяє відстежувати ці стани та підтримувати постійний зв'язок з клієнтом. Кожен новий запит обробляється окремо, без прив'язки до попередніх. Справитися з цією проблемою допомагає сесія браузера (сеанс) – механізм, який дозволяє відстежити запити від одного браузера та зберегти деякі зміни під час переходів по сторінках сайту.

З початком сесії на стороні сервера створюється файл, який містить інформацію про користувача, про його дії та події, що відбулися в рамках одного сеансу [6]. Такими подіями можуть бути перегляд сторінок сайту, різні взаємодії користувача з елементами сторінки, вчинення транзакцій і т. ін.

До тих пір, поки попередня сесія активна, нова не може бути розпочата. Старий сеанс може завершитися при дотриманні однієї з розумів (в залежності від налаштувань):

- після закінчення певної години бездіяльності (таймаут);
- у певний час доби (опівночі, наприклад);

— при закритті браузера.

1.10 Автентифікація: сесії vs токени

У сучасних веб-застосунках використовуються два основні підходи до автентифікації користувачів: сесії на основі cookies та токени. Обидва вирішують ту саму задачу — відстеження авторизованого стану користувача між HTTP-запитами, але різняться за місцем зберігання стану та масштабованістю.

Класичний підхід — автентифікація на основі сесій (cookie-based): після успішного входу сервер створює запис в пам'яті або базі (session store) та повертає клієнту ідентифікатор сесії у файлі-куках. При кожному наступному запиті браузер передає цей ідентифікатор, а сервер віднаходить відповідний серверний запис сесії.

Альтернативний підхід — автентифікація за токенами (обирається в сучасних REST-API та SPA): після успішного входу сервер видає підписаний токен (зазвичай JWT — JSON Web Token), який клієнт зберігає у localStorage або в пам'яті додатку та додає до кожного наступного запиту. Сервер не зберігає жодного стану про сесію користувача — токен сам по собі містить її ідентифікатор та права доступу.

Структура JWT містить три частини, розділені крапкою: header (алгоритм підпису, наприклад, HS256), payload (корисне навантаження — ідентифікатор користувача, термін дії exp, роль тощо), signature (підпис сервера для верифікації). Клієнт передає токен у заголовок Authorization: Bearer <token>; сервер перевіряє підпис без звернення до бази даних.

Порівняння двох підходів: сесії зручні для монолітних застосунків та дозволяють примусовий логат сервером (просто видалити файл сесії), але вимагають стану на сервері та погано масштабуються. Токени не вимагають стану, легко розподіляються між мікросервісами, але їх не можна примусово відізнати до моменту закінчення єхнього терміну дії (без додаткового списку відкликаних токенів).

Для вибору підходу враховується масштаб застосунку: для класичних веб-сайтів з одним сервером та серверним рендерингом зручнішими виявляються

сесії; для REST-API, який обслуговує мобільні додатки та SPA-клієнти, переважають JWT-токени. У комп'ютерному практикумі № 2 (NestJS-API) реалізується саме JWT-автентифікація.

1.11 Питання та завдання для самоконтролю

1. Які переваги архітектури клієнт-сервер перед програмами, що безпосередньо встановлюються на комп'ютері користувача?
2. Яка програма на комп'ютері користувача зазвичай виступає в ролі клієнта?
3. Що таке API?
4. Назвіть найбільш розповсюджені веб-сервери.
5. В чому полягає суть технології активних серверних сторінок?
6. Опишіть роботу протоколу HTTPs.
7. Які програми відносяться до серверної частини веб-застосунку?
8. Що таке Frontend і Backend?
9. Для чого слугують Файли-куки?
10. Чим формат JSON відрізняється від XML і чому JSON став стандартом для REST API?
11. Яку роль виконує заголовок Content-Type у HTTP-відповіді?
12. Чим відрізняється автентифікація на основі сесій від автентифікації за токенами?
13. З яких трьох частин складається токен JWT і як вони верифікуються сервером?

Лекція 2. ТИПИ ВЕБ-САЙТІВ

План лекції:

1. Статичні сайти на зорі Інтернету
2. Технологія AJAX
3. Single Page Application
4. Повернення до Server Side Rendering
5. Static Site Generator

Ключові слова: *HTML, CSS, PHP, SEO, сервер, ajax.*

2.1 Статичні сайти на зорі Інтернету

На зорі Інтернету були лише статичні сторінки, нічого динамічно не генерувалося. Звичайні, заздалегідь створені, статичні HTML документи надсилалися клієнту.

Коли користувач заходив на веб-сайт, простий HTTP запит йшов на сервер і далі він відповідав розміткою, яка відображалася в браузері.

Потім з'явилася можливість використовувати динамічний рендеринг і будувати шаблони для розмітки, наприклад PHP.

Ці шаблони дозволяли заповнювати інформацією, що надсилається клієнту, HTML. Кожен запит HTTP проходив через серверну частину веб-сайту і збирав необхідні дані. Наприклад, завдяки цьому з'явилася можливість додавати такі елементи як імена користувачів, поточні дати, дані з бази даних та інші.

Це і був початковий сервер Side Rendering. Клієнт (браузер) запитував у сервера, необхідний для відображення, HTML, який був згенерований саме на сервері і далі відображався у браузері.

Дамо визначення основним термінам:

– **SPA** – Single Page Application. Односторінкова програма – це веб-додаток або веб-сайт, що використовує єдиний HTML-документ як оболонку для всіх веб-сторінок і організує взаємодію з користувачем через HTML, CSS, JavaScript, що динамічно підвантажуються, зазвичай за допомогою AJAX;

- **CSR** – Client Side Rendering. Рендеринг на клієнті – це рендеринг програми у браузері за допомогою DOM (Document Object Model);
- **SSR** – Server Side Rendering. Рендеринг на сервері – це рендеринг клієнтської частини програми на сервері;
- **SSG** – Static Site Generator. Статична генерація сайтів – це генерація всіх HTML сторінок програми в момент збирання.

2.2 Технологія AJAX

З появою такої технології, як AJAX (Asynchronous JavaScript and XML), з'явилася можливість отримувати дані асинхронно, без перезавантаження цілої сторінки.

З точки зору UX, це було величезним поліпшенням. Зменшилася кількість перезавантажень сторінок, тим самим почався рух в бік Client Side Rendering.

Для спрощення роботи з фронтом стали з'являтися різні бібліотеки і фреймворки. Однією з найпопулярніших бібліотек стала jQuery, але вона ще не була повноцінним інструментом для CSR.

2.3 Single Page Application

Потім на зміну прийшли React, Angular і Vue, завдяки яким переважна більшість розробників познайомилася з повноцінним рендерингом на клієнті. Ці три фреймворки використовують компонентний підхід і дозволяють розбивати розмітку на дрібні частини, що перевикористовуються, тепер вся генерація HTML відбувається саме на фронтенді. Бекенд використовується тільки для складних обчислень, роботи з базами даних, насичення фронтенду даними та багато іншого.

Завдяки можливості легко генерувати розмітку на стороні клієнта, SPA завоювали величезну популярність.

Якщо трохи заглибитися в деталі, то тепер сервер став надсилати клієнту практично порожній HTML, який містить лише базову розмітку, з яким-небудь порожнім тегом, наприклад:

```
<div> class = ' root ' > </div>
```

і єдиний файл зі скриптом, наприклад:

```
< script src = ' bundle. js '></ script >
```

всередині якого буде написано весь код для генерації розмітки, стилів та логіки за допомогою JavaScript.

На жаль, такий підхід також призвів до цілої низки потенційних проблем.

По-перше, виникла проблема з пошуковою оптимізацією. Оскільки пошукові роботи Google читають і індексують веб-сайти, необхідно віддавати інформацію про контент і розмітку з сервера, а при такому підході все генерується на клієнті. Тоді ще пошукові системи не були здатні обробити інформацію згенеровану таким чином. Все, що міг побачити робот це порожній кореневий HTML -тег.

Наразі ситуація дещо покращилася, оскільки багато пошукових роботів навчилися виконувати, необхідний для Client Side Рендерингу, JavaScript. Проте результат такої індексації все одно бажає кращого.

По-друге, потенційною проблемою є продуктивність. Оскільки для відображення сторінки в браузері потрібне виконання великої кількості JavaScript, то програма може неабияк "гальмувати". Особливо це помітно на старих мобільних пристроях.

Для вирішення цих проблем розробники знову повернулися до ідеї створення розмітки на сервері.

2.4 Повернення до Server Side Rendering

Однак, на відміну від старого підходу, коли розмітка генерувалася на сервері за допомогою серверних мов програмування, наприклад PHP, тепер у Server Side Rendering, як і в CSR, будуть використовуватися сучасні JavaScript бібліотеки, наприклад React.

Різниця тільки в тому, що програма генеруватиме розмітку за допомогою React не на клієнтській стороні, а на серверній.

Це рішення було спрямоване на виправлення існуючих проблем з продуктивністю та SEO (Search Engine Optimization). Щоразу, коли пошуковий

робот запитує сторінку, він отримає необхідний контент – для візуалізації більше не потрібне виконання JavaScript на клієнті.

Тепер, коли у розробці згадують термін Server Side Rendering, то посилаються саме на цей сценарій.

Більше детально про Server Side Rendering (SSR).

Розглянемо плюси та мінуси SSR.

Плюси:

- SEO. Можливість налаштувати пошукову оптимізацію;
- First Contentful Paint. Завдяки Server Side Rendering, сторінки вашого сайту швидше стають доступними для взаємодії. Маючи продуктивний сервер, SSR може дати вам чудову оцінку First Contentful Paint, Що покращує UX і, можливо, також SEO сторінки.

Насправді це всі плюси, тепер розглянемо мінуси:

- уповільнюється час переходу між сторінками. Так, для початкового рендеру SSR швидше, ніж CSR, але далі при роботі з додатком вам доводиться малювати дані двічі, один раз на сервері та один раз на клієнті;
- більш складна технологія. Чистий клієнтський React значно простіший за стек з SSR. Відповідно збільшується кількість коду, складність розробки та з'являються додаткові бібліотеки;
- кешування. Для часткового закриття першого мінуса та на додаток до другого, додається більш складне кешування даних;
- вартість. SSR вимагає продуктивного сервера, що збільшує витрати на інфраструктуру порівняно з CSR.

Чотири основні моделі рендерингу серверного веб-стеку — CSR (клієнтський), SSR (серверний), SSG (статична генерація) та SPA (single-page application) — мають різні сильні та слабкі сторони. Нижче коротко зведено основні характеристики:

Таблиця 2.1 – Моделі рендерингу серверного веб-стеку

Критерій	CSR	SSR	SSG	SPA
Де відбувається рендер	Браузер	Сервер на запит	Build-time (заздалегідь)	Браузер після SPA-shell

продовження таблиці 2.1

First Contentful Paint	Повільний	Швидкий	Найшвидший	Повільний
SEO «з коробки»	Слабкий	Добрий	Добрий	Слабкий
Інтерактивність	Висока	Висока (+hydration)	Низька (статика)	Висока
Динамічні дані	Так	Так	Тільки build-time	Так
Серверні ресурси	Мінімальні	Високі	Майже нульові	Мінімальні
Типові випадки	Адмінки, дашборди	Контентні сайти, e-commerce	Блоги, документація, лендинги	Web-додатки (Gmail-style)
Приклад інструмента 2026	React, Vue	Next.js 14+ SSR mode	Next.js 14+ static export, Astro	React SPA, Vue SPA

У промисловій практиці 2026 року популярний гібридний підхід — універсальний рендеринг (Next.js 14+, Nuxt 3, SvelteKit), коли різні маршрути одного й того ж застосунку рендеряться у CSR / SSR / SSG залежно від характеру контенту: статичні лендинги — SSG, користувацький дашборд — CSR, продуктові сторінки з SEO — SSR. Шаблон Backend-for-Frontend (BFF) — окремий тонкий бекенд, що агрегує дані з мікросервісів спеціально під потреби фронтенду — добре поєднується з SSR.

Виявляється, у SSR величезна кількість мінусів, то що робити? Розглянемо ще один підхід — SSG.

2.5 Static Site Generator

SSG — це новий підхід для веб-розробки, який дозволяє створювати веб-сайти, сторінки яких генеруються в статичні файли в момент складання. Тим самим, коли браузер робить запит, сервер не генерує розмітку, як у випадку SSR, а віддає вже готову.

Різниця порівняно з класичним SSR в тому, що замість використання HTML і CSS тепер використовуються сучасні інструменти, такі як React, Vue і Angular. Тобто додаток, написаний на сучасних інструментах, перетворюється на HTML, CSS і JavaScript за допомогою транспіляторів, збирачів пакетів і т.д.

Проблема в тому, що SSG не передбачає динамічну роботу з сервером, тобто ви не можете створити, наприклад, кошик інтернет-магазину за допомогою SSG, оскільки інформація, що відображається в кошику, повинна бути унікальна

для кожного користувача. У такому випадку ми можемо застосувати лише SSR або CSR. А статична генерація сайтів зручна, наприклад, для сторінки "Доставка" або картки товару, тобто. для інформації, яка ідентична кожному за користувача.

Щоб детальніше зрозуміти інструмент, розглянемо його плюси та мінуси.

Плюси :

- SEO. Як і у випадку з SSR, пошуковий робот отримує всю необхідну інформацію;
- продуктивність. SSG має найбільшу продуктивність проти SSR і CSR, оскільки всі дані вже згенеровані під час складання;
- дуже дешево. Використовуючи статичну генерацію сайтів, вам не потрібний сервер, достатньо завантажити код свого сайту на GitHub і зібрати його, наприклад, за допомогою Netlify. У такому разі вам доведеться заплатити лише за доменне ім'я;
- простота розробки. Якщо порівнювати з CSR, то SSG дещо складніше, оскільки потрібно розібратися в тонкощах, але загалом картинка слабо відрізняється від звичайної розробки;
- безпека. У вашого статичного сайту немає сервера, а значить у зловмисників немає можливості отримати доступ до вашої бази даних або панелі адміністратора.

Мінуси:

- немає панелі адміністратора. Плюс, як відсутність уразливостей, але мінус, тому що всі зміни на сайті вам доведеться робити у редакторі коду, потім робити пуш у репозиторій, а далі збирати та деплоїти ваш сайт;
- SSG запустити складніше, ніж WordPress або іншу CMS. При початковому налаштуванні більшість CMS надає величезну кількість підказок для розгортання сайту. У разі роботи зі статичним генератором сайтів вам доведеться вивчати документацію;
- відображення лише статичних даних. Ви не можете відобразити динамічні дані за допомогою SSG, відповідно у вас немає можливості зробити досить складний веб-додаток.

Добре, SSG звучить частково краще ніж SSR, але як тоді розробити складне веб-додаток з динамічними даними, використовуючи сучасні інструменти і не втративши в SEO ?

SSG + SSR як універсальний рендеринг

На щастя, є метод під назвою універсальний рендеринг, за допомогою якого ви можете отримати найкращий з усіх підходів:

- швидкі та плавні переходи між сторінками, як у CSR;
- можливість роботи з SEO;
- неймовірно швидкий First Contentful Paint;
- роботу з динамічними даними;
- зменшення витрат, пов'язаних із вмістом сервера.

Усі ці можливості надає, наприклад, фреймворк Next.js 14+ — він з коробки підтримує SSR, SSG та гібридний рендеринг на одній і тій самій кодовій базі React.

2.6 Питання та завдання для самоконтролю

1. Чим відрізняються підходи клієнтського рендерингу (CSR) та серверного рендерингу (SSR) сторінок?
2. Які JavaScript фреймворки забезпечують повноцінний CSR?
3. Які переваги і недоліки SSR?
4. Які переваги і недоліки CSR?
5. В чому відмінність SSG від SSR?

Лекція 3. ОГЛЯД МОВ ДЛЯ СЕРВЕРНИХ ЗАСТОСУНКІВ

План лекції:

1. Типи мов програмування
2. HTML та CSS
3. Мова програмування Python
4. Мова програмування JavaScript
5. Мова програмування PHP
6. Мова програмування Java
7. Мова програмування C#

Ключові слова: *HTML, CSS, PHP, Python, JavaScript, Java, C#.*

3.1 Типи мов програмування

Мови програмування постійно розвиваються – коли одна відмирає, на зміну їй приходить нова. Якщо розвиватися тією мовою, яка подобається, то звісно можна заробити багато грошей. Але краще покладатися на перевірені мови програмування, які зарекомендували себе протягом тривалого часу та які користуються великим попитом.

Кожна мова програмування відрізняється від інших. Хоча кожна мова програмування має свій унікальний синтаксис, спосіб її написання, виконання та компіляції може змінити все.

Багато розробників віддають перевагу роботі з певними типами мов. Також набагато легше перемикається між схожими мовами, тому перша мова програмування, яку ви вивчаєте, має велике значення.

Мови програмування можуть бути практично будь-якими, але вони часто найкраще підходять для розробки програмного забезпечення, оскільки їх можна використовувати на різних платформах вони поєднуються. Більшість мов програмування призначені для розробки програмного забезпечення, різних програм, які ви завантажуєте і запускаєте на своїх пристроях.

Скриптові мови часто інтерпретуються, що означає, що їх код виконується на льоту, а не проходить процес компіляції в програмі. Такі мови веброзробки часто є мовами сценаріїв.

Мови розмітки насправді не є мовами програмування, але вони використовуються для веброзробки. Це читабельні теги, які використовуються для форматування документа.

Мови програмування спеціалізуються на створенні вебсторінок, як зовнішніх, так і внутрішніх.

Інтерфейсні (або клієнтські) мови модифікують вебсторінку в браузері користувача. Наприклад, якщо ви натискаєте на щось на вебсторінці та створюєте анімацію, це буде зроблено за допомогою інтерфейсного програмування, такого як CSS, HTML5 і JavaScript.

Внутрішні (або серверні) мови модифікують вебсторінку на рівні сервера або програми. Наприклад, коли ви надсилаєте дані з форми або змінюєте щось у базі даних – це внутрішнє програмування.

Слід також зазначити, що багато людей просто використовують мови програмування для позначення всіх мов програмування в цілому. Мова сценаріїв – це спеціалізована мова програмування, але не всі мови програмування є мовами сценаріїв.

3.2 HTML та CSS

Хоча HTML і CSS технічно не є мовами програмування, сьогоднішні їхні більш довершені версії HTML5 і CSS3 є ідеальними відправними точками, якщо ви хочете стати розробником сайтів чи вебінтерфейсів.

Поєднання цих двох (мов розмітки гіпертексту) формує блоки будь-якого вебсайту: HTML – це структурує та вміст сторінки, а CSS – це дизайнерське стилізування та модифікація структуру вебсторінки.

HTML та CSS — обов'язкова основа для будь-якого фронтенд-розробника легкою для вивчення мовою розмітки, CSS є більш складною, але також не важкою для вивчення.

Застосування: інтерфейсна веброзробка.

Переваги:

- дуже легко освоїти, навіть для людей без досвіду програмування;
- його висока популярність дозволяє легко знайти безкоштовні ресурси;
- добре підтримується на всіх пристроях.

Недоліки:

- існують незначні, але все ж проблеми з кросбраузерністю.

3.3 Мова програмування Python

Сплеск популярності Python, здавалося, виник нізвідки, але вона охопила майже всі сфери розробки.

Зараз це друга за популярністю мова програмування на GitHub (після JavaScript).

Python може все: від серверної частини до програмного забезпечення для машинного навчання.

У ній є майже все, що можна побажати від мови програмування: універсальність, швидкість та ефективність. Крім того, вона надзвичайно легка у вивченні порівняно з іншими мовами програмування.

Застосування: WEB і розробка програмного забезпечення.

Переваги:

- її можна використовувати майже будь-де, від веб-додатків до розробки програмного забезпечення та розробки ігор;
- кросплатформенність;
- висока популярність означає багато ресурсів і ще більше робочих місць.

Недоліки:

- повільніша, ніж інші мови;
- з Python складніше перейти на інші мови.

3.4 Мова програмування JavaScript

Незважаючи на те, що HTML і CSS є обов'язковими для інтерфейсних веброзробників, JavaScript надзвичайно популярний. У той час як структури

HTML і стилі CSS, JavaScript додає до вебсторінки розширену функціональність на стороні клієнта. На відміну від HTML і CSS, JavaScript є реальною мовою програмування та сценаріїв.

Застосування: мова програмування сценаріїв веб-сторінок. Рідко використовується для розробки мобільного чи програмного забезпечення.

Переваги:

- найшвидший і найпростіший спосіб програмування клієнтських сценаріїв, які виконуються у браузері;
- дуже популярна мова програмування;
- широка підтримка різноманітних програм;
- вона практично працює на більшості сучасних вебсайтів. На сервері 78% Node.js-проектів 2025 року написані TypeScript — статично типізована надбудова над JS (Microsoft, Apache 2.0). Серверна частина курсу (Розділ 8) — на TypeScript + NestJS.

Недоліки:

- для тих, хто знає лише мови розмітки, навчання може бути важким;
- має проблеми з безпекою та кросбраузерною.

3.5 Мова програмування PHP

PHP є мовою великих проєктів, наприклад найпопулярнішої системи управління сайтом як WordPress – мабуть, напевно всі чули про неї. Колись вона розділила спільноту розробників, оскільки була застарілою та повільною, і багато хто досі вважає, що її вивчення не варте часу.

PHP пройшла серйозну модернізацію у лінійці 8.x: JIT-компілятор, enum, readonly, match, іменовані аргументи, атрибути замість docblock-анотацій, строга типізація. За даними W3Techs 2025 року, 76% перевірених вебсайтів використовують PHP — це найпоширеніша мова бекенду на публічному вебі.

Ясно одне: PHP — це потужний спосіб програмування програм на стороні сервера, і її легко освоїти порівняно з іншими мовами сценаріїв.

Її популярність серед програмістів-початківців і велика кількість проєктів з відкритим кодом, таких як WordPress, означають, що тут також є багато навчальних ресурсів.

Існують десятки популярних фреймворків PHP, які можуть зробити вашу роботу з PHP ще простішою. З появою PHP 8.0, PHP намагається еволюціонувати від чистої серверної мови сценаріїв до мови програмування загального призначення.

Застосування: серверний веб-скрипт.

Переваги:

- дуже легко навчитися;
- добре зарекомендувала себе у веб-розробці та часто зустрічається на веб-сайтах;
- сучасні версії відносно швидкі;
- легко знайти роботу розробника PHP.

Недоліки:

- популярність падає порівняно з такими новими мовами, як Python і JavaScript.

3.6 Мова програмування Java

Java виглядає як пращур усіх мов програмування, але насправді вона навіть не така стара, як C++.

Навіть якщо багато хто вважає її застарілою, вона все ще використовується в усьому світі та на всіх видах пристроїв.

Python ось-ось випередить її і загалом Java впаде в популярності, але Java точно не вмерла. Є тисячі вакансій розробників серверної частини Java, і попит залишається високим, що робить її надійним вибором.

Незважаючи на те, що Java є старішою мовою, популярність якої впала з роками, Java залишається головним претендентом на популярність.

Застосування: розробка програмного забезпечення/додатків, веб і мобільних пристроїв.

Переваги:

- кросплатформенна та універсальна;
- давня але ще популярна, незважаючи на свій вік;
- більша безпека.

Недоліки:

- вимагає високого рівня абстрактного мислення;
- популярність має тенденцію до спаду.

3.7 Мова програмування C#

C# відома як сучасна, більш універсальна версія мови C++. З одного боку, C# набагато легше вивчити. Вона простіша і менш складна, її можна використовувати для створення різноманітних програм. Вона також набагато краще підходить для веб-розробки, ніж C++. Вона досить популярна для розробки ігор і знаходиться в середині списку найбільш високооплачуваних мов.

Між C# та C++ обирають за задачами: C++ — коли потрібна сира потужність, C# — коли пріоритет на швидкості розробки великих корпоративних застосунків.

Застосування: переважно розробка програмного забезпечення.

Переваги:

- універсальна.

Недоліки:

- відносно складна для вивчення.

3.8 Питання та завдання для самоконтролю

1. Яка мова, на вашу думку, є найкращою для початківця у веб-розробці?
2. Які мови застосовуються у frontend-розробці?
3. Які мови застосовуються у backend-розробці?
4. Які мови, на вашу думку, є взаємозамінними, частково або повністю?

Лекція 4. СЕРВІСНА АРХІТЕКТУРА

План лекції:

1. Основні принципи мікросервісної архітектури
2. Переваги мікросервісної архітектури
3. Монолітна архітектура
4. Порівняння сервісного та монолітного підходів
5. Перехід від моноліту до мікросервісів
6. Контейнеризація як інструмент сервісної архітектури
7. Брокери повідомлень: асинхронна комунікація між сервісами

Ключові слова: архітектура, мікросервіс, моноліт, контейнер.

4.1 Основні принципи мікросервісної архітектури

Мікросервісна архітектура – це метод побудови програмного забезпечення, який передбачає розділення додатку на декілька незалежних, спеціалізованих сервісів. Цей підхід протиставляється традиційній монолітній архітектурі, де вся функціональність додатка знаходиться в одному великому блоку коду. Цей розділ розглядає основи побудови сучасного веб-додатку на базі мікросервісів та проводить порівняння сервісного та монолітного підходу (рис. 4.1).

Мікросервіси – це невеликі, автономні сервіси, які виконують специфічні функції додатка та спілкуються між собою за допомогою API (Application Programming Interface). Основні принципи мікросервісної архітектури включають:

- відповідальність за одну річ: кожен сервіс відповідає за одну функцію та забезпечує стабільність її виконання;
- гнучкість: сервіси можуть бути розроблені, розгорнуті та масштабовані незалежно один від одного;
- міжсервісна комунікація: сервіси спілкуються між собою за допомогою стандартизованих протоколів, таких як REST, gRPC або GraphQL.

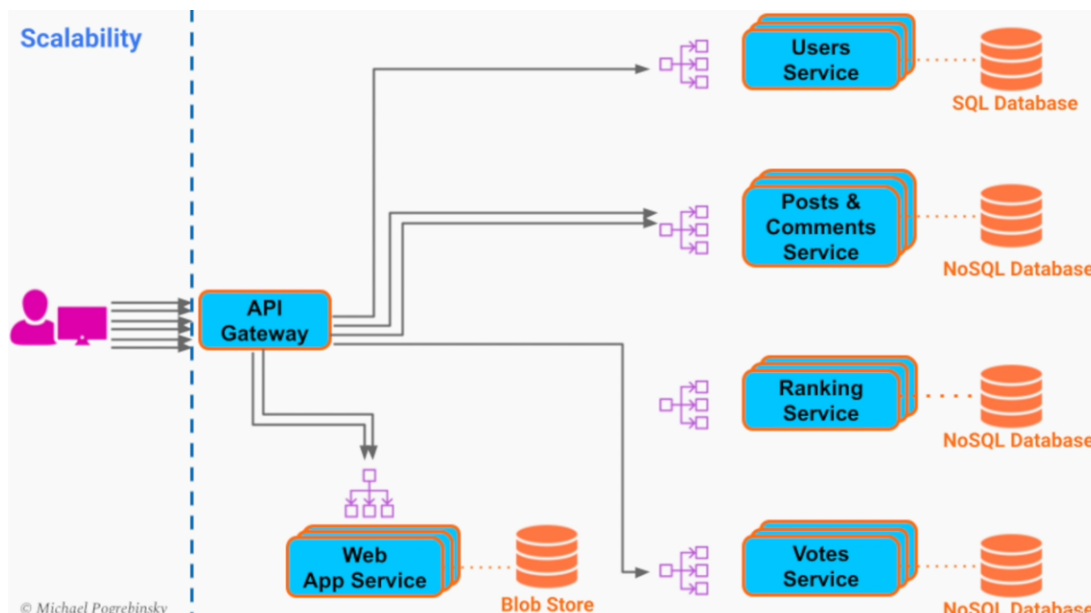


Рисунок 4.1 – Приклад мікросервісної архітектури застосунку

4.2 Переваги мікросервісної архітектури

Мікросервісна архітектура має декілька переваг порівняно з монолітним підходом:

- модульність: кожен сервіс може розроблятися, тестуватися та розгортатися незалежно від інших сервісів. Це полегшує відлагодження, розвиток та розширення функціональності додатка;
- масштабованість: сервіси можуть масштабуватися горизонтально, додаючи ресурси для окремих сервісів за потребами. Це дозволяє оптимізувати використання ресурсів та підвищити продуктивність системи;
- стійкість до відмов: у разі відмови одного сервісу, інші можуть продовжувати функціонувати, забезпечуючи певний рівень стійкості системи;
- технологічна гнучкість: різні сервіси можуть використовувати різні технології, мови програмування та бази даних, що дозволяє відібрати найбільш підходящі інструменти для кожного з них.

4.3 Монолітна архітектура

Монолітний підхід передбачає, що весь код додатка розміщується в одному блоку та ділить спільні ресурси (рис. 4.2). Основні характеристики монолітної архітектури:

- єдиний кодовий базис: усі компоненти додатка зберігаються в одному проєкті, що може ускладнювати розробку, тестування та розгортання;
- відсутність модульності: компоненти додатка тісно пов'язані, що може ускладнити відлагодження та модифікацію коду;
- вертикальна масштабованість: монолітні додатки можуть масштабуватися лише за допомогою додавання ресурсів до сервера, на якому вони розгорнуті.



Рисунок 4.2 – Приклад монолітної архітектури застосунку

Незважаючи на обмеження, монолітна архітектура має деякі переваги:

- простота розробки: монолітні додатки часто простіше розробляти та розгортати, оскільки вони мають єдиний кодовий базис та спільні ресурси;
- простота комунікації між компонентами: у монолітних додатках компоненти спілкуються безпосередньо, що може прискорити обмін даними та зменшити складність взаємодії між ними;
- менші витрати на інфраструктуру: монолітні додатки можуть вимагати менше ресурсів для розгортання, особливо на початкових етапах розвитку проєкту.

4.4 Порівняння сервісного та монолітного підходів

Вибір між мікросервісною та монолітною архітектурою залежить від потреб конкретного проєкту. Мікросервіси можуть бути вигіднішими для великих, складних додатків з великою кількістю функціональності та потребами в масштабованості. Монолітна архітектура може бути підходящою для невеликих проєктів з простою структурою та меншими вимогами до масштабування.

4.5 Перехід від моноліту до мікросервісів

Перехід від монолітної архітектури до мікросервісної може бути виконаний поетапно. Це може включати в себе винесення окремих функцій або компонентів монолітного додатка в окремі сервіси, реорганізацію коду та впровадження API для комунікації між сервісами.

Модульний моноліт

Модульний моноліт — компромісний архітектурний підхід між класичним монолітом та мікросервісами. Зовні це один процес, одне розгортання, одна БД (як у моноліті), але внутрішньо код суворо розбитий на модулі з чіткими публічними інтерфейсами; пряме звертання з одного модуля у внутрішні класи іншого забороняється на рівні архітектурних правил та інструментів (наприклад, `deptrac` у PHP).

Така структура зберігає простоту експлуатації моноліту (немає міжсервісної мережевої комунікації, складних транзакцій, розподілених трасувань) і одночасно формує чисті межі між доменами. Якщо у майбутньому модуль виросте до самостійного навантаження — його легко витягнути в окремий мікросервіс, бо залежності вже описані інтерфейсом.

`Symfony Bundles` — класичний приклад модульного моноліту: проєкт ділиться на `Bundle`-и (`UserBundle`, `BillingBundle`, `NotificationsBundle`), кожен бандл має свій контейнер сервісів, конфігурацію, міграції. `NestJS` пропонує таку ж модель через декоратор `@Module` — кореневий `AppModule` імпортує

функціональні модулі (UsersModule, AuthModule, OrdersModule), кожен зі своїми контролерами та сервісами. Деталі модульної архітектури NestJS — у Розділі 8.

4.6 Контейнеризація як інструмент сервісної архітектури

Контейнеризація (Docker, описаний у Розділі 9) органічно лягає на сервісну архітектуру за принципом «один контейнер — один сервіс». Кожен мікросервіс пакується у власний Docker-образ із власним середовищем виконання, версіями залежностей, конфігурацією; це усуває класичну проблему «у мене працює, у продакшні падає» — образ ідентичний на всіх етапах конвеєра.

Контейнери також полегшують поетапну міграцію від моноліту до сервісної архітектури: команда виносить за раз один обмежений домен у новий контейнер, залишаючи решту моноліту у старому. Docker Compose описує всю систему у декларативному файлі (див. 9.2.3), а оркестратори типу Kubernetes — масштабують сотні екземплярів у production. Третій комп'ютерний практикум курсу присвячено саме контейнеризації обох сервісів і налаштуванню CI/CD-конвеєра.

4.7 Брокери повідомлень: асинхронна комунікація між сервісами

Мікросервіси рідко спілкуються лише через HTTP-запити. Синхронний REST-виклик створює тісне зв'язування (coupling): сервіс-замовник блокується, поки сервіс-виконавець не відповість. Якщо ланцюжок включає 5 сервісів, загальна затримка дорівнює сумі затримок усіх викликів, а відмова будь-якого ланцюжка зриває весь сценарій. Для розвантаження таких сценаріїв і децентралізації відмов застосовують асинхронну комунікацію через брокер повідомлень.

Брокер повідомлень (Message Broker) — це посередник між сервісами. Сервіс-видавець (producer) надсилає повідомлення в брокер, а один або декілька споживачів (consumers) отримують його незалежно. Producer не знає, хто саме слухає, і не очікує негайної відповіді. Брокер забезпечує: збереження (persistence) повідомлень до моменту доставки, повторне надсилання у разі відмови

споживача, а також балансування навантаження між кількома consumer'ами. Це забезпечує decoupling (відсутність прямих знань про сусідні сервіси), resilience (відмова одного сервісу не зриває інших) і elasticity (черга поглинає піки навантаження).

Розрізняють два головні шаблони доставки. Queue (черга) — кожне повідомлення обробляє лише ОДИН consumer; типовий сценарій — фонові завдання (background jobs), де роботу треба виконати рівно один раз: відправлення e-mail, обробка завантаженого зображення, генерація PDF. Publish/Subscribe (Pub/Sub) — кожне повідомлення доставляється УСІМ підписникам топіка; сценарій — поширення подій між кількома незалежними сервісами: подія «замовлення створено» одночасно потрапляє в сервіс інвентаризації, сервіс розсилки та сервіс аналітики.

RabbitMQ — універсальний брокер на основі протоколу AMQP. Модель складається з елементів: exchange приймає повідомлення від producer'ів; queue зберігає повідомлення до отримання; binding визначає правило, за яким повідомлення з exchange потрапляє в queue (наприклад, за routing key). RabbitMQ підтримує гарантії at-least-once: повідомлення повторно надсилається, поки consumer не підтвердить його обробку через ACK. Сильні сторони — гнучкість routing (direct, topic, fanout, headers), підтримка протоколів AMQP/MQTT/STOMP, помірна складність налаштування. RabbitMQ — оптимальний вибір для робочих черг, RPC між сервісами та fan-out до кількох обробників. У Symfony стандартним інтерфейсом є компонент Messenger, у NestJS — модуль @nestjs/microservices.

Apache Kafka — лог-орієнтована платформа стрімінгу подій. Замість тимчасової черги Kafka зберігає повідомлення в розділених на партиції журналах (topics × partitions) тривалий час (дні, тижні чи місяці). Кожен consumer веде власний offset — позицію в журналі — і може повертатися назад або обробляти повідомлення повторно. Така архітектура ідеальна для event sourcing (відновлення стану системи з історії подій), потокової аналітики (Kafka Streams, ksqlDB), інтеграції з системами машинного навчання та обробки телеметрії. Kafka демонструє екстремальну пропускну здатність (мільйони повідомлень на

секунду), але вимагає більше зусиль у налаштуванні та оперуванні (кластер брокерів, ZooKeeper або KRaft, моніторинг).

Redis Pub/Sub — найпростіший варіант, побудований поверх Redis. Канали (channels) реалізують модель Publish/Subscribe з негайною доставкою (latency менша 1 мс). Однак Redis Pub/Sub НЕ зберігає повідомлень: якщо в момент публікації немає активних підписників, повідомлення губиться (at-most-once). Це робить його непридатним для гарантованої доставки, але оптимальним для real-time сценаріїв: миттєві нотифікації в UI, push-оновлення дашбордів, координація кешу між інстансами одного сервісу. Якщо проєкт уже використовує Redis для кешу, додавання Pub/Sub коштує одного рядка коду. Як компромісний варіант між Redis Pub/Sub і повноцінним Kafka — Redis Streams (з 5-ї версії Redis), що додає persistence та consumer groups.

Як обрати брокер для конкретного сценарію. Для робочих черг і фонових завдань оптимальний вибір — RabbitMQ: легкий старт, гнучкі шаблони доставки, гарантії at-least-once. Для інтеграційного шару, аналітики та event sourcing — Apache Kafka: висока пропускну здатність, тривале збереження подій, потокова обробка. Для real-time оновлень з мінімальною затримкою без вимог гарантованої доставки — Redis Pub/Sub або Redis Streams.

У великих системах нерідко поєднують усі три: Kafka для бізнес-подій, RabbitMQ для команд між сервісами, Redis Pub/Sub для UI-оновлень.

4.8 Підсумки до лекції

Сервісна архітектура, зокрема мікросервісна архітектура, пропонує ряд переваг у порівнянні з монолітною архітектурою, зокрема модульність, масштабованість, стійкість до відмов та технологічну гнучкість. Однак монолітна архітектура може бути підходящою для менших проєктів з простою структурою та меншими вимогами до масштабування.

При виборі архітектури для вашого проєкту рекомендується оцінити потреби та масштаби проєкту, складність функціональності, можливість розвитку та масштабування. Мікросервісна архітектура може виявитися вибором

для великих та складних проєктів, а монолітна архітектура – для невеликих та простих додатків.

У випадку потреби в переході від моноліту до мікросервісів, такий процес може бути проведений поетапно, з врахуванням аналізу поточної структури додатка та визначення стратегії переходу.

Загалом, вибір між мікросервісною та монолітною архітектурою залежить від специфіки кожного проєкту, його розвитку та технічних вимог. Важливо ретельно аналізувати потреби проєкту та враховувати переваги та недоліки кожного підходу, щоб прийняти оптимальне рішення.

4.9 Питання та завдання для самоконтролю

1. Що таке сервісна архітектура, та які основні принципи її роботи?
2. У чому полягають переваги мікросервісної архітектури порівняно з монолітною архітектурою?
3. Які основні характеристики та переваги монолітної архітектури?
4. Які фактори слід враховувати при виборі між мікросервісною та монолітною архітектурою для конкретного проєкту?
5. Які основні етапи переходу від монолітної архітектури до мікросервісної, та які аспекти слід враховувати під час такого переходу?
6. Що таке модульний моноліт і коли цей підхід є оптимальним компромісом між монолітною та мікросервісною архітектурами?
7. Як контейнеризація через Docker полегшує впровадження мікросервісної архітектури?
8. Які типові виклики виникають при міжсервісній комунікації у мікросервісній архітектурі (мережеві затримки, узгодженість даних, обробка помилок)?

Лекція 5. ОСНОВИ МОВИ PHP

План лекції:

1. Вступ
2. Основи мови PHP: змінні, константи, функції
3. Робота з даними: посилання, масиви, генератори
4. Чистий код: найменування, функції та форматування
5. Чистий код: класи, помилки та межі
6. Приклади коду PHP
7. Сучасний синтаксис PHP 8.x
8. Об'єктно-орієнтоване програмування у PHP
9. Практичні веб-приклади
10. Стандарти PSR

***Ключові слова:** PHP, змінні, константи, функції, масиви, класи, стиль програмування, форматування коду.*

5.1 Вступ

PHP — це скриптова мова програмування, створена для генерації HTML-сторінок на стороні вебсервера. Спочатку PHP розшифровувалася як **Personal Home Page**, але нині її офіційна назва **PHP: Hypertext Preprocessor**.

PHP є однією з найпоширеніших мов, яку використовують у сфері веброзробки, її підтримує більшість хостинг-провайдерів. PHP інтерпретує вебсервер у HTML-код, який передається на сторону клієнта. На відміну від JavaScript, користувач не бачить PHP-коду, тому що браузер отримує готовий HTML-код. Це є перевагою з погляду безпеки, але погіршує інтерактивність сторінок [5].

Активна сторінка PHP є HTML-сторінкою, в текст якої вставлені інструкції мови PHP. Інструкції знаходяться всередині обмежувачів `<?php... ?>`. Файли сторінок PHP, зазвичай, мають розширення `php`, `phtml` або `phpml`. Під час оброблення сторінки, що містить код PHP, лексичного аналізу й інтерпретації

піддається тільки вміст конструкції `<?php... ?>`. Решта всього текстового змісту (код розмітки) прямує у вихідний потік без зміни.

5.2 Основи мови PHP: змінні, константи, функції

Власне, код PHP складається з набору інструкцій. PHP створений не тільки для форматування статичного тексту. Для того, щоб обробляти різні дані були придумані змінні. Змінна - контейнер з даними. Кожна змінна містить певне значення.

Синтаксис змінної складається з знака долара - \$ і "вільного" ідентифікатора якому присвоюється якесь значення [3].

Змінна створюється тоді, коли їй присвоюють якесь значення. Для присвоєння значення змінної використовують оператор присвоєння, який складається з знака рівності (Змінну можна вивести на екран за допомогою оператора `echo`).

Як правило, назви змінних починаються з маленької літери або символу підкреслення. Варто враховувати, що PHP є регістрозалежною мовою, а значить, змінні `$num` і `$Num` будуть представляти дві різні змінні.

Також при іменуванні змінних нам треба враховувати такі правила:

1. імена змінних повинні починатися з алфавітного символу або підкреслення;
2. імена змінних можуть містити лише символи: `a-z`, `A-Z`, `0-9`, та знак підкреслення;
3. імена змінних не повинні включати прогалини.

Коли не потрібно міняти задане значення для змінної, то має сенс створити константу і потім використовувати її в будь-якій частині скрипта. Для опису константи використовують функцію `define`, якій передається її ім'я і значення [3]. Константи, як і змінні, зберігають певне значення, однак на відміну від змінних константа може набутися значення лише один раз, і далі ми можемо його змінити. Константи зазвичай визначаються для зберігання значень, які мають залишатися незмінними протягом усієї роботи скрипта.

Для визначення константи застосовується оператор `const`, у своїй назві константи знак долара \$ (на відміну змінних) не використовується. Зазвичай

назви констант використовують великі символи, але це умовність. Після визначення константи ми можемо її використовувати так само, як і звичайну змінну. PHP дозволяє встановлювати значення констант на основі обчислюваних виразів.

Також для визначення константи може застосовуватися функція `define()`. Параметр `$name` передає назву константи, а параметр `$value`— її значення. Значення константи може бути типом `int`, `float`, `string`, `bool`, `null` або масивом.

Магічні константи.

Крім створюваних програмістом констант у PHP є ще кілька так званих "магічних" констант, які є частиною мови за замовчуванням [3] (див. таблицю 5.1).

Таблиця 5.1 – Деякі константи PHP

Назва константи	Опис
<code>__FILE__</code>	зберігає повний шлях та ім'я поточного файлу
<code>__LINE__</code>	зберігає поточний номер рядка, який обробляє інтерпретатор
<code>__DIR__</code>	зберігає каталог поточного файлу
<code>__FUNCTION__</code>	назва оброблюваної функції
<code>__CLASS__</code>	назва поточного класу
<code>__TRAIT__</code>	назва поточного трейту
<code>__METHOD__</code>	назва оброблюваного методу
<code>__NAMESPACE__</code>	назва поточного простору імен

Щоб перевірити, чи визначено константу, ми можемо використовувати функцію `bool defined (string $name)`. Якщо константа `$name` визначена, то функція повертатиме значення `true`.

Функція — це блок коду, який може бути іменований і викликаний повторно. Іноді функцію називають підпрограмою. Ми звикли, що звичайній змінній можна присвоїти число, рядок або масив, а потім отримати його назад, звернувшись до значення на ім'я змінної.

Функції є блоком інструкцій, які багаторазово можна викликати в різних частинах програми. Функції дозволяють розділяти програму на менші функціональні частини. Визначення функції починається з ключового слова `function`, за яким слідує ім'я функції [5]. Ім'я функції має починатися з

алфавітного символу або підкреслення, за якими може йти будь-яка кількість алфавітно-цифрових символів або символів підкреслення. Після імені функції у дужках йде перелік параметрів. Навіть якщо параметрів функції немає, то просто йдуть порожні дужки. Потім у фігурних дужках йде тіло функції, що містить набір інструкцій.

Функції бувають вбудовані та користувальницькі. *Вбудовані функції* за вас вже написали автори мови, і ви можете просто брати їх і використовувати. У PHP існують тисячі готових функцій на всі випадки життя, наприклад, `sort()` для сортування масивів, `print()` для виведення рядків на екран або функції для роботи з базами даних.

Користувальницькі функції програмісти пишуть самі — наприклад, щоб перевірити дані пасажирів за номером авіаквитка або визначити, чи зараз високосний рік. Ці функції зазвичай використовуються тільки всередині одного проєкту, але бувають винятки — і такі функції виносять до бібліотек.

Аргументи функції — це змінні, які ми передаємо у функцію обробки. Аргументів може бути кілька.

Правильне і зворотне - змінні, визначені всередині функції, не будуть доступні ззовні. Такі змінні називаються локальними, оскільки вони локальні стосовно функції. На відміну від аргументів, яких може бути кілька, повернути у зовнішній код функція може лише одне значення — за допомогою інструкції «`return`» (повернення). Значення, що повертається називають результатом роботи функції.

5.3 Робота з даними: посилання, масиви, генератори

Посилання в PHP дозволяють посилатися на область пам'яті, де розташовано значення змінної або параметра. Для створення посилання перед змінною вказується символ амперсанда & [3].

Що таке посилання на PHP? Посилання в PHP - це засіб доступу до вмісту однієї змінної під різними іменами. Вони не схожі на покажчики C; наприклад, ви не можете виконувати над ними адресну арифметику, вони не є реальними адресами пам'яті і т.д.

Масив – це набір даних, які об'єднані під одним ім'ям. Масив складається з кількох елементів, які мають певний індекс [3].

Масиви створюються з допомогою оператора присвоєння, як і змінна. Імена масивів починаються зі знака \$, після якого слідує довільний ідентифікатор, далі йдуть квадратні дужки: `$arr[0] = "php";`

Дана конструкція створює масив і надає його елементу з індексом 0 значення "php", після чого ми можемо звертатися до цього елемента як до звичайної змінної:

```
echo $arr[0].
```

В результаті ми побачимо слово *php*.

Генератори надають легкий спосіб реалізації простих ітераторів без використання додаткових ресурсів чи складнощів, пов'язаних із реалізацією класу, що реалізує інтерфейс *Iterator*. Генератор дозволяє вам писати код, що використовує `foreach` для перебору набору даних без необхідності створення масиву в пам'яті, що може призвести до перевищення ліміту пам'яті, або зажадає багато часу для його створення. Замість цього, ви можете написати функцію-генератор, яка, по суті, є звичайною функцією, за винятком того, що замість повернення єдиного значення, генератор може повертати (`yield`) стільки разів, скільки необхідно для генерації значень, що дозволяють перебрати вихідний набір даних. Наочним прикладом вищесказаного може бути використання функції `range()` як генератора. Стандартна функція `range()` генерує масив, що складається із значень, і повертає його, що може призвести до генерації величезних масивів даних. Наприклад, виклик `range(0, 1000000)` призведе до використання понад 100 МБ оперативної пам'яті. В якості альтернативи ми можемо створити генератор `xrange()`, який використовує пам'ять тільки для створення об'єкта *Iterator* і збереження поточного стану, що вимагатиме не більше 1 кілобайта пам'яті [3].

Коли функція генератор викликається, вона поверне об'єкт вбудованого класу *Generator*. Цей об'єкт реалізує інтерфейс *Iterator* стане односпрямованим об'єктом ітератора і надасть методи, за допомогою яких можна керувати його станом, включаючи передачу в нього і повернення з нього значень.

5.4 Чистий код: найменування, функції та форматування

Універсального рецепту чистого коду не існує — є той підхід, що найкраще підходить для конкретного завдання. Під час написання коду варто відповідати собі на питання [4]:

- які кейси можуть мати завдання?
- чи все я врахував?
- що може йти не так?
- що можна поєднати?
- чи є схожий функціонал?
- що тут зайве?
- як зробити простіше?
- як зробити зрозуміліше?

Чистий код простий, виразний, читається легко як проза. Він не повинен бути жорстко зчепленим — кожна сутність легко замінюється або перевикористовується. Правило бойскаута: «Залиш місце стоянки чистішим, ніж воно було до тебе» [4]. Бачиш «брудний код» — зроби його трохи чистішим у межах своєї задачі; якщо забруднення системне, винось окремий рефакторинг у беклог. Будь-яка сутність має відповідати за один функціонал і лише за нього. І вона має виконувати його добре — Single Responsibility. Якщо сутність відповідає одночасно двом і більше діям, її функціонал треба розділяти. Код повинен читатись зверху вниз. У гарній та грамотній архітектурі внесення змін обходиться без значних витрат. Видаляй "мертвий код". "Мертвий код" — це код, який не буде викликаний за жодних умов або код, який ніде не використовується.

Використовуй зрозумілі і зручні імена для будь-яких сутностей. Вони повинні описувати, чому ця сутність існує, що вона робить і як використовується. Довжину імен змінних, методів варто намагатися робити не надто довгою. Ідеальною довжиною вважається 8 символів. Але якщо зміст ім'я не відображає зміст змінної, функціонал методу, призначення класу чи

інтерфейсу, варто збільшити довжину. Довге та зрозуміле ім'я краще, ніж коротке, але незрозуміле. Якщо назва сутності не відповідає її функціоналу або за назвою не зрозуміло, що сутність робить, то її треба перейменувати на найзрозумілішу назву. Якщо цього зробити неможливо, то значить із її функціоналом щось не так і її треба рефакторити. Сутність, яка має назву “And”, “With” — порушує Single Responsibility. Функціонал такої сутності варто поділяти. Незрозумілі тексти, рядки варто виносити у змінні та давати їм зрозумілі назви. Назви методів повинні містити дієслово, яке описує, що цей метод робить. Потрібно уникати однакових найменувань для двох різних цілей [4].

Якщо сутність має схожу з іншою сутністю назву, то швидше за все їх функціонал дуже схожий і їх потрібно об'єднати. Якщо ні, їх назви потрібно змінювати, щоб вони не співпадали. Якщо ти подумки перейменовуєш сутність, коли читаєш код, щоб тобі було зрозуміліше розуміти її функціонал, то перейменуй її в цю назву. Виберіть одне слово для однієї концепції. Важко буде розуміти функціонал, коли в тебе є `fetch`, `retrieve` та `get` у назвах. Нехай краще скрізь буде `get`.

Функції мають бути короткими та компактними. Оптимальним розміром вважається 20 рядків і 150 символів в одному рядку, якщо не влізть, потрібно розділяти. Функція повинна виконувати лише одну операцію. Вона повинна виконувати її добре і нічого іншого вона не повинна робити. Якщо функція виконує лише ті дії, які знаходяться на одному рівні абстракції, функція виконує одну операцію. Щоб визначити, чи виконує функція більше однієї операції, спробуй витягти з неї іншу функцію, яка не буде простою переформулюванням реалізації. Будь-які умовні оператори з довгими виборами через `switch-case`, `if-else` повинні розділятися або об'єднуватися без дублювання, можливо на класи з реалізаціями, а вибір реалізації передати базовому класу, фабриці чи будь-яким іншим чином. `If`, `else`, `while` і т.д. повинні містити виклик однієї функції [6]. Так буде читабельніше, зрозуміліше та простіше. Якщо вхідних аргументів більше трьох, то варто задуматися, яким чином краще їх позбутися, наприклад, створити клас для цих аргументів. Що більше вхідних аргументів, то важче розуміється

функція. Функція, в яку передається аргумент-прапор, від якого залежить робота функції свідчить, що функція виконує більше однієї операції. Такі функції слід розбити на дві та викликати їх рівнем вище. Функція, яка змінює вхідний аргумент, має віддавати посилання на змінений об'єкт, а не просто змінювати без повернення.

```
String transform(String text)
```

Якщо функція повинна змінювати вхідний аргумент, то нехай вона змінює стан свого об'єкта-власника. Якщо вхідний аргумент функції не повинен змінюватися (і використовується далі в коді), слід скопіювати значення аргументу і всередині функції працювати з копією. Замість `return null` краще використовувати порожній об'єкт `Collection.empty()` або `null`-об'єкт - `EmptyObject()`. Якщо є код, який повинен слідувати один за одним, передай результати першої функції в другу, щоб хтось не змінив послідовність викликів. Використовуй поліморфізм замість `if/else` чи `switch/case` чи `when`. Уникай негативних умов.

Не використовуй коментарі, якщо ти можеш використовувати функцію або змінну замість цього. Не коментуй поганий код – перепиши його. Не варто пояснювати, що відбувається у поганому коді, краще зробити його явним та зрозумілим. Коментарі можна використовувати при передачі будь-якої інформації, попередження про наслідки, але з пояснення того, як працює код. Використовуй `TODO` та `FIXME` у тих випадках, коли потрібно помітити, що код потребує доопрацювання, але зараз немає ресурсів на це [4].

Документуй код, який є складним, але чистим. Не залишай старий закоментований код. Ти можеш знайти його в історії комітів, якщо потрібно. Коментарі мають бути короткими та зрозумілими. У коментарях з інформацією не має бути багато інформації. Все має бути стисло і змістовно.

Дотримуйся `codestyle`, прийнятий на проєкті. Дотримуйся правил, прийнятих у команді. Сталий `codestyle` робить код передбачуваним для кожного у команді. Потрібно мати автоматичні засоби, які форматуватимуть код за тебе. Структура файлу має бути зрозумілою з першого погляду: вгорі — заголовок та параметри, далі — функції від верхнього рівня абстракції до нижнього. Сутності, пов'язані один з одним, повинні бути поруч, наприклад, в одному пакеті, щоб

було простіше здійснювати навігацію за кодом. Змінні (поля) класу повинні бути вгорі класу. Методи повинні знаходитися ближче до свого місця використання. Функції повинні знаходитись у порядку виклику. Якщо одна викликає іншу, то функція, що викликає, повинна перебувати над викликаногою. З іншого боку, приватні функції нижчого рівня можуть бути внизу файлу і заважати розумінню коду високого рівня. Необхідно працювати з абстракціями, щоб реалізацію можна було легко змінити. Клієнт, який використовує функціонал, не повинен знати про деталі реалізації, він повинен знати, яку реалізацію в якому разі використовувати. Необхідно надавати API, з яким варто працювати та приховувати деталі реалізації, структуру. Так буде простіше працювати з сутностями та додавати нові види поведінок, функціоналу та реалізацій [6].

DTO - Data Transfer Object. Клас, який містить лише дані та ніякого функціоналу. Потрібний для того, щоб передавати якісь дані. Об'єкт такого класу має бути незмінним.

5.5 Чистий код: класи, помилки та межі

Класи мають бути компактними. Ім'я класу має описувати його відповідальність. Функціонал класу повинен чітко відповідати та вписуватися в назву класу. Розділяй пов'язаність на маленькі класи. Жорсткої та рясної зв'язаності не повинно бути— це ускладнює підтримку та розвиток проєкту [6].

Пам'ятай про Single Responsibility. Сутність повинна мати одну і лише одну причину зміни. Дотримуйся інкапсуляції. Ослаблення інкапсуляції завжди має бути останньою мірою. Зазвичай ми оголошуємо змінні та допоміжні функції приватними, але іноді їх потрібно оголошувати `protected` та мати змогу звернутися до неї з тесту. Якщо група функцій належить до певного функціоналу, то цю групу функцій можна виділити в окремий клас і використовувати його екземпляр.

Використовуйте Exceptions замість повернення кодів помилок. Обробка помилок – це одна операція. Якщо у функції є ключове слово `try`, то після блоків `catch/finally` нічого іншого у функції не повинно бути. Якщо в тебе є `enum`, який перераховує помилки, то його краще позбутися і замість нього використовувати

винятки. Використовуйте `unchecked exceptions`, щоб явно вказати на місце в якому є проблеми. Такі помилки не потрібно відловлювати, натомість потрібно написати код так, щоб цієї помилки ніколи не було. Передавай достатню кількість інформації разом із появою виключення, щоб потім користувачі твого коду могли зрозуміти, що справді сталося. Замість умовних операторів із обробкою помилок краще викидати винятки та обробляти їх. Не передавай `null` будь-куди. Намагайся цього максимально уникнути. Обробка помилок – це окреме завдання і не відноситься до основної логіки програми [6].

Ми завжди використовуємо будь-які бібліотеки, які найчастіше дають нам занадто широкий функціонал або конфліктують із очікуваним функціоналом, що робить код бруднішим у його кінцевому використанні. Уникнути цього можна просто, застосувавши патерни типу Decorator, Adapter, Facade або інші. Бувають ситуації, коли тобі потрібно працювати з функціоналом, який знаходиться у розробці або поки що не адаптований для використання у коді продакшен. У цьому випадку варто уявити, чого ти чекаєш від бібліотеки/цього функціоналу і написати свій інтерфейс або створити сутність з якими ти працюватимеш у своєму проєкті так, як тобі потрібно. Коли бібліотека доробиться та стане стабільною, ти адаптуєш її під свої готові структури та використати вже готовий функціонал.

5.6 Приклади коду PHP

Приклад № 1 Перший скрипт на PHP: hello.php

Створіть файл з ім'ям `hello.php` у кореневому каталозі веб-сервера (`DOCUMENT_ROOT`) і запишіть у нього наступне:

```
<html>
  <head>
    <title>Тестуємо PHP</title>
  </head>
  <body>
    <?php echo '<p>Привіт, світ!</p>'; ?>
  </body>
</html>
```

Відкрийте файл у браузері, набравши ім'я вашого веб-сервера та `/hello.php`. При локальній розробці це посилання може бути чимось на зразок

`http://localhost/hello.php` або `http://127.0.0.1/hello.php`, але це залежить від налаштувань вашого сервера. Якщо все налаштовано правильно, файл буде оброблений PHP і браузер виведе наступний текст:

```
<html>
  <head>
    <title> Тестуємо PHP </title>
  </head>
  <body>
    <p>Привіт, світ!</p>
  </body>
</html>
```

Ця програма надзвичайно проста, і для створення настільки простої сторінки навіть необов'язково використовувати PHP. Все, що вона робить, це виведення Hello World, використовуючи інструкцію PHP `echo`. Зверніть увагу, що файл не повинен бути виконуваним, або ще якось відрізнитися від інших файлів. Сервер знає, що цей файл повинен бути оброблений PHP, тому що файл має розширення `".php"`, про яке в налаштуваннях сервера сказано, що подібні файли повинні передаватися PHP. Розглядайте його як звичайний HTML-файл, якому пощастило отримати набір спеціальних тегів (доступних також і вам), здатних на купу цікавих речей.

Якщо у вас цей приклад не відображає нічого або виводить вікно завантаження, або якщо ви бачите весь цей файл у текстовому вигляді, то, швидше за все, ваш веб-сервер не має підтримки PHP або налаштований неправильно. Переконайтеся, що ви запитуєте файл на сервері через протокол `http`. Якщо ви просто відкриєте файл із вашої файлової системи, він не буде оброблений PHP. Мета прикладу – показати формат спеціальних тегів PHP. У цьому прикладі ми використовували `<? php` як тег, що відкривається, потім йшли команди PHP, що завершуються закриваючим тегом `?>`. Таким чином можна будь-де "застрибувати" і "вистрибувати" з режиму PHP в HTML файлі.

Приклад № 2 Отримання інформації про систему з PHP

```
<?php phpinfo(); ?>
```

Приклад № 3 Виведення значення змінної (елемента масиву)

Визначимо, який браузер використовує той, хто дивиться зараз нашу сторінку. Для цього ми перевіримо рядок з ім'ям браузера, що надсилається нам

у HTTP-запиті. Ця інформація зберігається у змінній. Змінні в PHP завжди передуються знаком долара. Змінна, що цікавить нас в даний момент, називається `$_SERVER['HTTP_USER_AGENT']`.

```
<?php
echo $_SERVER['HTTP_USER_AGENT'];
?>
```

Приклад виведення цієї програми:

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)

У PHP існує безліч типів змінних. У попередньому прикладі ми виводили елемент масиву.

`$_SERVER` – це одна із змінних, які надаються вам мовою PHP. Список таких змінних можна переглянути в розділі "Зарезервовані змінні " або переглянувши виведення функції `phpinfo()`, яка використовується у попередньому прикладі.

Всередині PHP-тегів можна поміщати кілька виразів і створювати маленькі блоки коду, що роблять більше, ніж простий виклик `echo`. Наприклад, якщо ви хочете додати перевірку для Internet Explorer.

Приклад № 4 Приклад використання керуючих структур та функцій

```
<?php
if (strpos($_SERVER['HTTP_USER_AGENT'], 'MSIE') !== FALSE) {
    echo 'Ви використовуєте Internet Explorer.<br/>';
}
?>
```

Приклад виведення цієї програми:

Ви використовуєте Internet Explorer.

Тут ми показали кілька нових елементів. По-перше, тут є конструкція `if`. Якщо ви знайомі з основами синтаксису мови C, ви вже помітили щось схоже. Якщо ж ви не знаєте C або подібної за синтаксисом мови, то найкращий варіант - взяти якусь вступну книжку з PHP і прочитати перші пару розділів.

Крім цього, тут є виклик функції `strpos()`. `strpos()` - вбудована в PHP функція, яка шукає один рядок в іншому. У цьому випадку ми шукаємо рядок 'MSIE' (так звану голку - needle) в `$_SERVER['HTTP_USER_AGENT']` (у так

званому "сіні" - haystack). Якщо "голка" знайдена всередині "сіна", функція повертає позицію "голки" щодо початку "сіна". Інакше вона повертає false. Якщо вона не поверне false, то умова в if виявиться істинною (true), і код у фігурних дужках ({ }) виконається. В іншому випадку код не виконується. Якщо ви не знаєте, як використовувати функції, можливо, вам варто прочитати сторінку посібника про те, як читати визначення функцій та розділ про функції PHP.

Приклад № 5 Змішування режимів HTML та PHP

Продемонструємо, як можна входити в режим PHP коду і виходити з нього навіть прямо посередині блоку з кодом.

```
<?php
if (strpos($_SERVER['HTTP_USER_AGENT'], 'MSIE') !== FALSE) {
?>
    <h3>strpos() повернув не false</h3>
    <p>Ви використовуєте Internet Explorer</p>
    <?php
} else {
?>
    <h3>strpos() вернул false</h3>
    <p>Ви не використовуєте Internet Explorer</p>
    <?php
}
?>
```

Приклад виведення цієї програми:

```
<h3>strpos() повернув не false</h3>
<p>Ви використовуєте Internet Explorer</p>
```

Замість використання команди PHP echo для виведення ми вийшли з режиму коду і просто послали вміст HTML. Важливий момент тут те, що логічна структура PHP коду при цьому не втрачається. Тільки одна HTML-частина буде надіслана клієнту залежно від результату функції strpos() (іншими словами, залежно від того, знайдено рядок MSIE чи ні).

Одна з найголовніших переваг PHP – те, як він працює з формами HTML. Тут основним є те, що кожен елемент форми автоматично стає доступним вашим програмам на PHP.

Приклад № 6 Найпростіша форма HTML

Приклад форми HTML:

```
<form action="action.php" method="post">
  <p>Ваше ім'я: <input type="text" name="name" /></p>
  <p>Ваш вік: <input type="text" name="age" /></p>
```

```
<p><input type="submit" /></p>
</form>
```

У цій формі немає нічого особливого. Це звичайна форма HTML без спеціальних тегів. Коли користувач заповнить форму та натисне кнопку відправки, буде викликана сторінка `action.php`.

Приклад № 7 Виводимо дані форми

У файлі `action.php` може бути наступне:

```
Вітаю, <?php echo htmlspecialchars($_POST['name']); ?>.
Вам <?php echo (int)$_POST['age']; ?> років.
```

Приклад виведення цієї програми:

```
Вітаю, Сергію. Вам тридцять років.
```

Якщо не брати до уваги фрагменти коду з `htmlspecialchars()` і `(int)`, принцип роботи цього коду повинен бути простим і зрозумілим. `htmlspecialchars()` забезпечує правильне кодування "особливих" HTML-символів так, щоб шкідливий HTML або JavaScript не був вставлений на вашу сторінку. Поле `age`, про яке нам відомо, що воно має бути число, ми можемо просто перетворити на `int`, що автоматично позбавить нас небажаних символів. PHP також може зробити це автоматично за допомогою модуля `filter`. Змінні `$_POST['name']` та `$_POST['age']` автоматично встановлені для вас засобами PHP. Раніше ми використовували суперглобальну змінну `$_SERVER`, тут ми так само використовуємо суперглобальну змінну `$_POST`, яка містить всі POST-дані. Зауважимо, що метод відправлення (`method`) нашої форми – POST. Якби ми використовували метод GET, то інформація нашої форми була б у суперглобальній змінній `$_GET`. Крім цього, можна використовувати змінну `$_REQUEST`, якщо джерело даних не має значення. Ця змінна містить суміш GET, POST, COOKIE.

5.7 Сучасний синтаксис PHP 8.x

PHP 8.x — це сучасна, типобезпечна мова з чистим синтаксисом, що принципово відрізняється від PHP 5/7 минулих років. Лінійка 8 (8.0 у листопаді 2020, 8.3 у листопаді 2023) додала JIT-компілятор, перерахування, конструктор-

промоушен, readonly-властивості, іменовані аргументи, атрибути замість docblock-анотацій та вираз match замість громіздкого switch. Цей підрозділ показує те, що ви будете писати щодня у Symfony та Doctrine.

Жорстка типізація

За замовчуванням PHP виконує неявне приведення типів: функція з оголошенням `int` може отримати рядок «42» і працюватиме. У продакшні це джерело важко виловлюваних помилок. Директива `declare(strict_types=1)` робить типи строгими — невідповідність типу спричиняє `TypeError` під час виклику.

```
<?php
declare(strict_types=1); // ОБОВ'ЯЗКОВО ставити першим рядком
файлу

function priceWithVat(int $price, float $vatRate = 0.20): int {
    return (int) round($price * (1 + $vatRate));
}

echo priceWithVat(1000);           // 1200, OK
echo priceWithVat('1000');        // TypeError у строгому режимі
echo priceWithVat(price: 1000);    // 1200 — іменований аргумент
```

Параметри типізуються анотацією перед іменем (`int $price`), повертальний тип — після `:`. У сучасному коді практично всі функції та методи мають типи на вході та виході — це робить рефакторинг безпечним і документує API.

Перерахування (enum)

До PHP 8.1 фіксований набір значень доводилося представляти константами класу — це не давало типового контролю. Enum усуває цю проблему: оголошений набір значень — це окремий тип, який не можна сплутати з рядком чи числом.

```
enum OrderStatus: string
{
    case Draft      = 'draft';
    case Confirmed  = 'confirmed';
    case Shipped     = 'shipped';
    case Delivered   = 'delivered';
    case Cancelled   = 'cancelled';

    public function isFinal(): bool {
        return match($this) {
            self::Delivered, self::Cancelled => true,
            default => false,
        };
    }
}
```

```

    }
}

function processOrder(Order $order, OrderStatus $newStatus): void
{
    if ($order->status->isFinal()) {
        throw new LogicException('Замовлення вже закрито');
    }
    $order->status = $newStatus;
}

// З БД повертається рядок — конвертуємо в enum:
$status = OrderStatus::from('shipped'); //
OrderStatus::Shipped
$status = OrderStatus::tryFrom('foo'); // null (без винятку)

```

«Backed enum» (з оголошеним типом string або int) дозволяє зберігати перерахування у БД як рядок чи число й легко перетворювати назад через `from()` / `tryFrom()`. Doctrine 3+ нативно підтримує enum-типи у сутностях, тож кодуйте бізнес-стан перерахуванням, а не магічним рядком.

Вираз match

`match` — це сучасний аналог `switch` без падіння (fall-through), без необхідності `break`, з підтримкою декількох умов в одному рукаві та з обов'язковим `default` (або винятком `UnhandledMatchError` при невідповідності). На відміну від `switch`, `match` — вираз, тобто повертає значення.

```

// Старий стиль (switch — багатослівно й помилкоємно):
switch ($httpCode) {
    case 200:
    case 201:
    case 204:
        $message = 'OK';
        break;
    case 404:
        $message = 'Not Found';
        break;
    default:
        $message = 'Other';
}

// Сучасний стиль (match — лаконічно й безпечно):
$message = match($httpCode) {
    200, 201, 204 => 'OK',
    400           => 'Bad Request',
    404           => 'Not Found',
    422           => 'Unprocessable Entity',
    500, 502, 503 => 'Server Error',
    default      => 'Unknown',
}

```

```
};
```

Readonly та конструктор-промоушен

PHP 8.0 додав конструктор-промоушен — оголошення властивостей прямо у параметрах конструктора. PHP 8.1 додав readonly — властивості, які встановлюються один раз (зазвичай у конструкторі) і потім не можуть бути змінені. Поєднання дає лаконічні і безпечні DTO та value-objects:

```
// До PHP 8.0 — багатослівно:
class ProductDto {
    public string $title;
    public float $price;
    public function __construct(string $title, float $price) {
        $this->title = $title;
        $this->price = $price;
    }
}

// PHP 8.0 — конструктор-промоушен:
class ProductDto {
    public function __construct(
        public string $title,
        public float $price,
    ) {}
}

// PHP 8.1+ — readonly: незмінні після ініціалізації
class ProductDto {
    public function __construct(
        public readonly string $title,
        public readonly float $price,
    ) {}
}

$dto = new ProductDto('Ноутбук', 24990.50);
$dto->title = 'Інше'; // Error: Cannot modify readonly property
```

Іменовані аргументи

Іменовані аргументи (PHP 8.0+) дозволяють передавати значення за іменем параметра, а не позицією. Це особливо корисно для функцій з багатьма опціональними параметрами:

```
function sendEmail(
    string $to,
    string $subject,
    string $body,
    bool $html = false,
    array $cc = [],
    array $bcc = [],
    int $priority = 3,
```

```

): bool { /*... */ }

// Без іменованих аргументів — нечитабельно й помилкоємно:
sendEmail('user@kpi.ua', 'Тема', '<p>Hello</p>', true, [], [], 5);

// З іменованими аргументами — самодокументований код:
sendEmail(
    to:      'user@kpi.ua',
    subject: 'Тема',
    body:    '<p>Hello</p>',
    html:    true,
    priority: 5,
);

```

Атрибути

Атрибути (PHP 8.0+) — нативний механізм метаданих, що замінює docblock-анотації (@Route, @ORM\Entity тощо). Атрибути — це звичайні класи з декоратором #[Attribute]; їх читають через рефлексію. У Symfony 7 та Doctrine 3 атрибути — стандарт; docblock-анотації офіційно застаріли.

```

// Старий стиль (Doctrine 2 + анотації у docblock):
/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 */
class Product { /*... */ }

// Сучасний стиль (PHP 8 атрибути):
#[ORM\Entity]
#[ORM\Table(name: 'products')]
class Product { /*... */ }

```

5.8 Об'єктно-орієнтоване програмування у PHP

PHP підтримує повноцінне ООП з PHP 5 (2004). У сучасному коді (Symfony, Doctrine, будь-який бекенд-фреймворк) ви будете створювати десятки класів на проєкт. Цей підрозділ повторює базові концепції з акцентом на ідіоми PHP 8.x.

Класи та об'єкти

```

class User
{
    public function __construct(
        public readonly string $email,
        public readonly string $name,
        private string $passwordHash,
    ) {}

    public function verifyPassword(string $plain): bool {
        return password_verify($plain, $this->passwordHash);
    }
}

```

```

    }

    public function changePassword(string $newPlain): void {
        $this->passwordHash = password_hash($newPlain,
PASSWORD_DEFAULT);
    }
}

$user = new User(
    email: 'admin@kpi.ua',
    name: 'Олександр Бунке',
    passwordHash: password_hash('secret123', PASSWORD_DEFAULT),
);

```

Видимість властивостей: `public` (доступ ззовні), `private` (тільки в межах класу), `protected` (клас + спадкоємці). Сучасна практика — приховувати дані за `private/protected`, відкривати методи доступу (геттери/сеттери) лише там, де це справді потрібно. `readonly + public` — поширений спосіб створити `value-object` без сеттерів.

Інтерфейси

Інтерфейс — контракт, що описує набір методів без реалізації. Класи, які реалізують інтерфейс, зобов'язані надати ці методи. Інтерфейси — основа поліморфізму та принципу залежності від абстракцій (D з SOLID).

```

interface PaymentGateway
{
    public function charge(int $amountCents, string $currency):
string;
    public function refund(string $transactionId): bool;
}

class StripeGateway implements PaymentGateway
{
    public function charge(int $amountCents, string $currency):
string {
        //... виклик Stripe API
        return $stripeChargeId;
    }
    public function refund(string $transactionId): bool { /*... */
}
}

class LiqPayGateway implements PaymentGateway
{
    public function charge(int $amountCents, string $currency):
string { /*... */ }
    public function refund(string $transactionId): bool { /*... */
}
}

```

```

}

// Замовник (CheckoutService) залежить від інтерфейсу, а не від
// конкретної реалізації
class CheckoutService
{
    public function __construct(private PaymentGateway $gateway)
    {}
}

```

Абстрактні класи

Абстрактний клас — клас, який не можна інстанціювати безпосередньо; він призначений для успадкування. Може містити часткову реалізацію (на відміну від інтерфейсу) та абстрактні методи, які зобов'язаний реалізувати спадкоємець.

```

abstract class Notification
{
    public function __construct(protected string $recipient) {}

    abstract protected function buildBody(): string;
    abstract public function send(): bool;

    // Спільна логіка для всіх типів сповіщень
    protected function logAttempt(): void {
        error_log("Notification to {$this->recipient}");
    }
}

class EmailNotification extends Notification
{
    protected function buildBody(): string { /* HTML */ return '...'; }
    public function send(): bool {
        $this->logAttempt();
        return mail($this->recipient, 'Subject', $this->buildBody());
    }
}

```

Трейти

PHP — мова з одиничним успадкуванням (extends лише один клас). Якщо потрібно поділитися кодом між кількома непов'язаними класами — використовуйте трейти. Трейт — це шматок коду, який «вшивається» в клас на етапі компіляції.

```

trait Timestampable
{
    private \DateTimeImmutable $createdAt;
    private ?\DateTimeImmutable $updatedAt = null;

    // Викликати з конструктора класу, що використовує трейт

```

```

private function initTimestamps(): void {
    $this->createdAt = new \DateTimeImmutable();
}

public function getCreatedAt(): \DateTimeImmutable {
    return $this->createdAt;
}

public function touch(): void {
    $this->updatedAt = new \DateTimeImmutable();
}
}

class Product {
    use Timestampable;

    public function __construct(public readonly string $title) {
        $this->initTimestamps();
    }
}

class User {
    use Timestampable;

    public function __construct(public readonly string $email) {
        $this->initTimestamps();
    }
}

```

Властивість `$createdAt` оголошена з типом `\DateTimeImmutable` без значення за замовчуванням і БЕЗ ?-нульованості. У PHP 8 строго типізовані властивості обов'язково мають бути ініціалізовані до першого читання — інакше «Typed property must not be accessed before initialization». Тому клас, що використовує трейт, мусить покликати `initTimestamps()` у своєму конструкторі. Альтернатива — зробити властивість `?\DateTimeImmutable` з `default null`, але для `timestamp` у сутності це гірше дидактично: кожне читання потребуватиме перевірки на `null`.

Простори імен (namespaces)

Простір імен — спосіб організувати класи, інтерфейси, функції та константи в ієрархічну структуру, аналогічно каталогам у файловій системі. У реальних проєктах простори імен ВІДПОВІДАЮТЬ структурі каталогів — це і є PSR-4 автозавантаження (див. підрозділ 5.D).

```

// src/Service/Email/Mailer.php
namespace App\Service\Email;

use App\Entity\User;
use Psr\Log\LoggerInterface;

```

```
class Mailer
{
    public function __construct(private LoggerInterface $logger)
    {}
    public function sendWelcome(User $user): void { /*... */ }
}
```

Інструкція `use` імпортує клас з іншого простору імен у поточний — після цього можна використовувати коротке ім'я (`User`, `LoggerInterface`) замість повного (`\App\Entity\User`). Імена класів за конвенцією `PascalCase`, простори імен — теж `PascalCase` з роздільником `\` між рівнями.

Обробка винятків

Сучасний PHP-код повідомляє про помилкові ситуації виключно через викидання винятків (exceptions), а не через коди повернення чи прапорці. Виняток — об'єкт, що зупиняє нормальне виконання й «спливає» по стеку викликів, поки не буде перехоплений у блоці `catch`. Базовий клас — `\Throwable`, практично всі винятки спадкуються від `\Exception`.

```
use App\Service\PaymentGateway;
use App\Exception\InsufficientFundsException;
use App\Exception\PaymentDeclinedException;
use Psr\Log\LoggerInterface;

// Власні класи винятків — рекомендована практика
class InsufficientFundsException extends \Exception {
    public function __construct(public readonly int $available,
    public readonly int $required) {
        parent::__construct("Available $available, required
    $required");
    }
}

function checkout(Order $order, PaymentGateway $gateway,
    LoggerInterface $logger): void {
    try {
        $gateway->charge($order->totalCents, 'UAH');
    } catch (InsufficientFundsException $e) {
        $logger->warning('Insufficient funds', ['needed' => $e-
        >required]);
        throw new CheckoutFailedException('Не вистачає коштів',
        previous: $e);
    } catch (PaymentDeclinedException $e) {
        // інша обробка
    } finally {
        $logger->info('Checkout attempt finished');
    }
}
```

Блок `finally` виконується завжди — і при успіху, і при винятку, і навіть якщо в `catch` його перевикинули. Туди типово виносять закриття ресурсів, відписку тощо. Параметр `previous`: `$e` зберігає ланцюг винятків — при логуванні видно первинну причину.

5.9 Практичні веб-приклад

Цей підрозділ показує три щоденні задачі веб-розробника на сучасному PHP: робота з JSON, маніпуляції масивами через функціональний стиль (`array_map` / `array_filter` / `array_reduce`) та виконання HTTP-запитів засобами стандартної бібліотеки. Усі приклади розраховані на PHP 8.x.

Робота з JSON

JSON — стандартний формат обміну даними у REST API. PHP має дві базові функції: `json_encode` (PHP-об'єкт або масив → JSON-рядок) та `json_decode` (JSON-рядок → масив або об'єкт). У сучасному коді обов'язково використовуйте прапорець `JSON_THROW_ON_ERROR` — інакше помилки парсингу мовчазно повернуть `null`.

```
// Кодування у JSON
$data = [
    'id' => 42,
    'title' => 'Ноутбук',
    'price' => 24990.50,
    'tags' => ['electronics', 'sale'],
];
$json = json_encode($data, JSON_THROW_ON_ERROR |
    JSON_UNESCAPED_UNICODE);

// Декодування з JSON: 2-й параметр true → асоціативний масив
try {
    $array = json_decode($json, associative: true, flags:
    JSON_THROW_ON_ERROR);
} catch (\JsonException $e) {
    error_log('Невалідний JSON: ' . $e->getMessage());
    throw new BadRequestException('Невалідний формат запиту',
    previous: $e);
}
```

Прапорець `JSON_UNESCAPED_UNICODE` зберігає кириличні символи у вигляді «Ноутбук», а не «`\u041d\u043e\u0443\u0442\u0431\u0443\u043a`» — компактніше й читабельніше. У Symfony JSON-серіалізація вже виконується серйознішим Symfony Serializer, але ці базові функції залишаються корисними для скриптів та простих ендпоінтів.

Сучасні масиви: функціональний стиль

Замість циклів `for/foreach` з ручним накопиченням результату сучасний PHP-код часто використовує функції `array_map`, `array_filter`, `array_reduce`. Це короткіше, виразніше, безпечніше — функції не мутують вхідний масив.

```
$products = [/* список товарів-об'єктів */];

// 1. array_filter — лишити тільки ті, що наявні та дешевші
30000 грн
$scheap = array_filter(
    $products,
    fn(Product $p) => $p->inStock && $p->price < 30000,
);

// 2. array_map — отримати масив назв
$titles = array_map(fn(Product $p) => $p->title, $scheap);

// 3. array_reduce — підрахувати загальну вартість
$total = array_reduce(
    $scheap,
    fn(float $sum, Product $p) => $sum + $p->price,
    0.0,
);

// 4. Spread-оператор — об'єднання масивів (PHP 7.4+)
$first = ['a', 'b', 'c'];
$second = ['d', 'e'];
$combined = [...$first, ...$second, 'f']; //
['a','b','c','d','e','f']

// 5. Деструктуризація з ключами (PHP 7.1+)
$user = ['name' => 'Alex', 'age' => 30, 'city' => 'Kyiv'];
['name' => $name, 'city' => $city] = $user;
```

HTTP-запит засобами PHP

Найпростіший спосіб виконати HTTP-запит — `file_get_contents` зі `stream-context`. Для серйозного коду рекомендується PSR-18 HTTP-клієнт (Guzzle, Symfony HttpClient), але базовий синтаксис теж корисно знати:

```
// GET-запит
$response = file_get_contents('https://api.example.com/products');
$products = json_decode($response, true, flags:
JSON_THROW_ON_ERROR);

// POST-запит з тілом JSON
$context = stream_context_create([
    'http' => [
        'method' => 'POST',
        'header' => "Content-Type: application/json\r\n"
            . "Authorization: Bearer secret-token\r\n",
```

```

        'content' => json_encode(['title' => 'Новий товар',
'price' => 999]),
        'timeout' => 5,
    ],
);
$response = file_get_contents('https://api.example.com/products',
false, $context);

```

Зверніть увагу: виконання HTTP-запитів через `file_get_contents` вимагає, щоб у налаштуваннях `php.ini` була увімкнена директива `allow_url_fopen`. На production-серверах її часто вимикають з міркувань безпеки (захист від SSRF — підставних URL-схем у вхідних даних, які перетворили б ваш код на шлюз до внутрішньої мережі). Якщо `allow_url_fopen=0`, код видасть warning і поверне false — тоді єдиний шлях це cURL (`php-curl-extension`) або готовий HTTP-клієнт.

У реальному Symfony-кодi HTTP-запити виконуються через ін'єкований `HttpClientInterface` (`Symfony HttpClient`) — він підтримує асинхронні запити, перерви, автоматичну серіалізацію та інтегрується з логуванням і профайлером. Файлові потоки залишають для коротких скриптів.

5.10 Стандарти PSR

PSR (PHP Standards Recommendations) — стандарти, прийняті спільнотою PHP-FIG (Framework Interop Group). Це угоди між фреймворками про спільні інтерфейси, правила автозавантаження та стиль коду. PSR не нав'язує реалізацію — він забезпечує сумісність: компонент, написаний за PSR-15, працює і в Symfony, і в Laravel, і в Slim. Найважливіші для нас — PSR-4 та PSR-12.

PSR-4: автозавантаження

PSR-4 описує однозначне правило: повне ім'я класу `App\Service>Email\Mailer` має жити у файлі `src/Service/Email/Mailer.php`. Composer автоматично будує карту цих відповідностей за конфігурацією у `composer.json` і генерує автозавантажувач — більше не потрібно явно викликати `require/include` для кожного класу.

```

// composer.json
{
    "autoload": {
        "psr-4": {

```

```

        "App\\": "src/",
        "Tests\\": "tests/"
    }
}

// src/Entity/User.php — простір імен ОВОВ'ЯЗКОВО відповідає шляху
namespace App\Entity;
class User { /*... */ }

// public/index.php — підключення лише одного автозавантажувача
require __DIR__ . '/../vendor/autoload.php';
$user = new App\Entity\User(); // автоматично знайдено в
src/Entity/User.php

```

Після зміни `composer.json` (або додавання нового класу) виконайте `composer dump-autoload` — Composer перебудує карту класів. У Symfony це робиться автоматично при `composer install/update`. Symfony's PSR-4 структура — стандарт для всіх нових PHP-проєктів.

PER Coding Style — стиль коду PHP

Стандарт PSR-12, який раніше визначав стиль PHP-коду, у 2023 році офіційно переведений PHP-FIG у статус «лише для довідки» — він не встигав за швидким розвитком синтаксису PHP 8 (атрибути, `readonly`, `match`, перерахування). На зміну прийшов PER Coding Style (PER-CS) — пряме еволюційне продовження PSR-12, що регулярно оновлюється та відображає сучасні правила. Інструменти `PHP_CodeSniffer` та `PHP-CS-Fixer` мають готовий стандарт PER (PER-CS2.0 на момент 2026 року).

Ключові правила PER-CS (зворотно сумісні з PSR-12):

1. Файли в кодуванні UTF-8 без BOM, з Unix-перенесенням рядків (`\n`).
2. Відступ — 4 пробіли (НЕ табуляція).
3. Простір після коми, навколо бінарних операторів: `$a + $b`, `foo($x, $y)`.
4. Відкриваюча фігурна дужка класу/функції — на новому рядку;
`if/else/while` — на тому ж рядку.
5. Імена класів — `PascalCase`, методи й змінні — `camelCase`, константи — `UPPER_SNAKE_CASE`.
6. Один клас на файл; ім'я файлу збігається з ім'ям класу.
7. Атрибути PHP 8 — на окремих рядках перед оголошенням, кожен зі своєю парою [...].

8. Promoted constructor parameters — кожен на окремому рядку для читабельності.

```
// PER-CS-сумісний код:
namespace App\Service;

use App\Entity\User;
use Psr\Log\LoggerInterface;

class UserService
{
    public const MAX_LOGIN_ATTEMPTS = 5;

    public function __construct(
        private LoggerInterface $logger,
    ) {}

    public function register(string $email, string $password):
    User
    {
        if (strlen($password) < 8) {
            throw new \InvalidArgumentException('Пароль
закороткий');
        }
        $this->logger->info('Registering user', ['email' =>
$email]);
        return new User($email, password_hash($password,
PASSWORD_DEFAULT));
    }
}
```

Перевірити та виправити стиль коду автоматично можна інструментом PHP_CodeSniffer: `vendor/bin/phpcs --standard=PER src/` показує порушення; `vendor/bin/phpcbf --standard=PER src/` виправляє більшість з них автоматично. Альтернатива — PHP-CS-Fixer (`vendor/bin/php-cs-fixer fix --rules=@PER-CS2.0 src/`), його часто інтегрують у CI/CD-конвеєр як обов'язковий лінтер перед мерджем PR.

Інші важливі PSR (для ознайомлення)

1. PSR-3 — інтерфейс `LoggerInterface`; кожен фреймворк використовує його (наявно в `Monolog`).
2. PSR-7 — представлення HTTP-повідомлень (`Request`, `Response`) як об'єктів.
3. PSR-11 — інтерфейс `ContainerInterface` для DI-контейнера; усі сучасні фреймворки відповідають.
4. PSR-15 — `middleware`-конвеєр для HTTP-обробки.

5. PSR-18 — інтерфейс HTTP-клієнта.

Питання для самоконтролю до підрозділу 5.D: Що таке PSR і хто їх розробляє? Чим PER Coding Style відрізняється від PSR-12 і чому останній більше не підтримується? За яким правилом простір імен класу `App\Service\Mailer` відповідає його файлу? Як Composer знає, де шукати класи? Що таке `composer dump-autoload` і коли його запускати? Які ключові правила PER-CS щодо відступів та фігурних дужок? Чим PSR-3 (`LoggerInterface`) корисний для портабельності коду?

5.11 Питання та завдання для самоконтролю

1. Сформулюйте переваги мови PHP для створення динамічних сторінок.
2. Яким чином PHP-скрипти розміщуються на веб-сторінках?
3. Сформулюйте відмінності в синтаксисі PHP і знайомих вам мов програмування.
4. Назвіть способи отримання доступу до даних із форм у PHP.
5. Які ви знаєте вбудовані константи PHP?
6. Які правила іменування змінних в PHP?
7. Що таке генератори в PHP?
8. Що таке масив у PHP?
9. Яким чином дані змінних можна вставляти в рядки?
10. Що відбувається в результаті деструктуризації масиву?
11. Що таке фігурний синтаксис?
12. Чим оператор `match` у PHP 8 відрізняється від `switch` і коли краще обрати `match`?
13. Що таке `readonly`-властивість і чому вона зручна для незмінних об'єктів-значень?
14. Які переваги дає просування властивостей через конструктор (`constructor property promotion`) у PHP 8?
15. Що таке PHP-атрибути (`Attributes`) і як їх використовують `Symfony` та `Doctrine`?

16. Чим стандарт PER Coding Style відрізняється від PSR-12 і чому він прийшов йому на заміну?

Лекція 6. ПАКЕТНИЙ МЕНЕДЖЕР COMPOSER

План лекції:

1. Вступ
2. Інсталяція Composer
3. Встановлення на Linux/Unix/macOS
4. Встановлення на Windows
5. Синтаксис та опції Composer
6. Структура composer.json та composer.lock
7. Семантичне позначення версій
8. Оновлення пакетів
9. Видалення пакетів
10. Скидання автозавантаження

Ключові слова: *composer, контроль версій, синтаксис.*

6.1 Вступ

Розробки мовою програмування PHP швидко розвиваються кожного року. Щомісяця реєструють десятки, або навіть сотні бібліотек для роботи з PHP проектами. В PHP для керування бібліотеками використовують пакетний менеджер Composer. Це потужний і зручний інструмент, він легко справляється з усіма завданнями щодо встановлення та вирішення залежностей у бібліотеках. Цей інструмент дозволяє не тільки встановлювати сторонні пакети, але й оновлювати їх при виході нових версій. Також за допомогою Composer можна легко створювати пакети для бібліотек.

6.2 Інсталяція Composer

Composer можна встановити локально, в директорії проєкту, або глобально. Встановивши Composer глобально, його можна використовувати у будь-якому проєкті.

Детальну інструкцію із завантаження інсталятора можна прочитати за посиланням: <https://getcomposer.org/download/>. Після завантаження інсталяційного файлу можна розпочинати інсталяцію Composer.

6.3 Встановлення на Linux/Unix/macOS

Локально

Після правильного завантаження та встановлення інсталятора у директорії проєкту з'явиться файл 'composer.phar'.

Щоб запустити Composer локально, достатньо виконати команду в терміналі, перебуваючи у директорії проєкту.

```
php composer.phar
```

Глобально

Для того щоб зробити Composer глобальним і викликати його з будь-якої директорії, достатньо виконати команду в терміналі:

```
mv composer.phar /usr/local/bin/composer
```

Тепер запустіть:

```
composer
```

6.4 Встановлення на Windows

Для того щоб встановити Composer глобально для Windows, потрібно виконати наступні кроки:

Створити новий composer.bat файл поруч з composer.phar:

Використовуючи cmd.exe:

```
C:\bin> echo @php "%~dp0composer.phar" %*>composer.bat
```

Використовуючи PowerShell:

```
PS C:\bin> Set-Content composer.bat '@php "%~dp0composer.phar" %*'
```

Додайте каталог до змінного середовища PATH, якщо це ще не зроблено. Для отримання інформації про зміну змінної PATH дивіться статтю за посиланням: <https://www.computerhope.com/issues/ch000549.htm>.

Закрийте поточний термінал. У новому вікні терміналу введіть наступну команду, щоб перевірити роботу Composer:

```
C:\Users\username>composer -V
Composer version 2.0.12 2021-04-01 10:14:59
```

6.5 Синтаксис та опції Composer

Перше, що необхідно сказати, Composer — це консольна утиліта, вона не має графічного інтерфейсу, однак це не робить її гіршою. Її синтаксис досить простий, а подивитися всі опції та команди можна ввівши команду 'composer' у терміналі.

Список команд, що часто використовуються:

- `require <package>` – встановити пакет як залежність і додати запис у `composer.json`
- `install` – встановити всі залежності з `composer.json` (точні версії беруться з `composer.lock`)
- `create-project` – створює проєкт на основі пакета в зазначену директорію
- `remove <package>` – видалити пакет із залежностей та оновити `composer.lock`
- `dump-autoload` – оновлює систему автозавантаження класів
- `update` – оновлює всі пакети до найактуальнішої версії

Встановлення проєкту на основі пакету

Швидше за все, для майбутніх проєктів, ви використовуватимете фреймворки, щоб мати базовий функціонал з коробки. У таких випадках вам знадобиться розгортати проєкт на основі існуючого пакета, Composer завантажить пакет з репозиторію і просто розпакує його в потрібну вам директорію.

Для цього використовуйте команду **create-project**, наприклад:

```
composer create-project laravel/laravel app_dir
```

де:

- `composer` – звернення до утиліти;
- `create-project` – команда;
- `laravel/laravel` – пакет фреймворку laravel;
- `app_dir` – директорія, в яку Composer розпакує вказаний пакет.

Встановлення пакетів

Щоб інсталиувати пакет за допомогою Composer, необхідно використовувати команду **require**. Утиліта встановить вказаний пакет і запише його у файл `composer.json`.

Наприклад, встановимо пакет для зручного виведення змінних:

```
composer require larapack/dd
```

Також можна вказати опцію `--dev` перед написанням назви пакета, щоб пакет був обов'язковим, але використовувався лише у розробці.

Усі доступні пакети можна переглянути в сервісі [Packagist](https://packagist.org).

6.6 Структура `composer.json` та `composer.lock`

Файл `composer.json` — маніфест PHP-проєкту: метадані (ім'я, версія, опис, автори), залежності виконання (`require`) та розробки (`require-dev`), конфігурація автозавантаження (`autoload`, `autoload-dev`) та допоміжні скрипти. Створюється командою `composer init` та автоматично оновлюється при `composer require/remove`.

Приклад `composer.json` для Symfony-проєкту:

```
{
  "name": "kpi/products-api",
  "description": "REST API на Symfony 7 для каталогу продукції",
  "type": "project",
  "license": "MIT",
  "require": {
    "php": ">=8.2",
    "symfony/framework-bundle": "^7.0",
    "doctrine/orm": "^3.0",
    "doctrine/doctrine-bundle": "^2.11"
  },
  "require-dev": {
    "phpunit/phpunit": "^10.5",
    "symfony/maker-bundle": "^1.50"
  },
  "autoload": {
    "psr-4": { "App\\": "src/" }
  },
  "autoload-dev": {
    "psr-4": { "App\\Tests\\": "tests/" }
  }
}
```

`composer.lock` — автоматично згенерований файл з ТОЧНИМИ версіями всіх встановлених пакетів (включно з транзитивними залежностями).

Створюється Composer'ом при `composer install / require / update`. КОМІТЯТЬСЯ обидва файли (`composer.json` + `composer.lock`) — це гарантує, що команда розробників, CI та production отримують ОДНАКОВИЙ набір версій. `composer install` читає `composer.lock`; `composer update` — ігнорує `lock` та оновлює залежності у межах обмежень `composer.json`, перезаписуючи `lock`-файл.

6.7 Семантичне позначення версій

SemVer (Semantic Versioning, <https://semver.org>) — стандарт нумерації версій у форматі MAJOR.MINOR.PATCH (наприклад, 7.0.4):

- 1) MAJOR — несумісні зміни API; оновлення вимагає міграції коду.
- 2) MINOR — нова функціональність зі зворотною сумісністю.
- 3) PATCH — виправлення помилок зі зворотною сумісністю.

У `composer.json` обмеження версії описують, які саме оновлення дозволені.

Найважливіші оператори:

```
"symfony/framework-bundle": "^7.0"      // ^ caret: 7.x.x, але НЕ 7.0.0
"phpunit/phpunit":           "^10.5"    // 10.x.x з 10.5 і вище
"doctrine/orm":              "~3.1"      // ~ tilde: 3.1.x, але НЕ 3.2.0
"some/legacy-package":       "1.2.3"     // exact: рівно 1.2.3
"some/dev-package":          "*"          // будь-яка версія (НЕ рекомендовано)
"some/range-package":        ">=2.0,<3.0" // явний діапазон
```

Найпоширеніший і рекомендований оператор — caret (^) з вказанням MINOR: `^7.0` дозволяє автоматично отримувати оновлення в межах MAJOR=7 (тобто всі MINOR і PATCH-релізи лінії 7.x), але блокує перехід на 8.0, де очікуються breaking changes. Це баланс між автоматичними виправленнями безпеки і контрольованими міграціями: розробник свідомо приймає рішення про MAJOR-оновлення ($^7 \rightarrow ^8$).

Що НЕ варто робити: ставити зірочку ("*") у production-залежностях — це означає згоду на будь-яку версію, включно з breaking-релізами; ставити exact-версію без crucial причин — блокує отримання критичних виправлень безпеки.

6.8 Оновлення пакетів

Пакети необхідно оновлювати, щоб уникнути уразливостей у проєктах та усунення старих проблем з боку підключених бібліотек.

Для оновлення всіх пакетів достатньо ввести команду **update**:

```
composer update
або
composer update -dev
```

6.9 Видалення пакетів

Для видалення пакета необхідно ввести команду **remove** та вказати потрібний пакет, також можна додати опцію `-dev`, якщо це пакет для розробників.

```
composer remove larapack/dd
```

Можна також видалити непотрібні пакети у файлі `composer.json` та запустити команду на оновлення.

6.10 Скидання автозавантаження

Composer з коробки може бути ролі автозавантажувача класів. Він використовує стандарт PSR-4. Іноді трапляється, що класи закешувалися, і новий клас не помітний у проєкті.

У таких випадках необхідно виконати команду скидання:

```
composer dump-autoload
або
composer du
```

6.11 Питання та завдання для самоконтролю

1. Що таке Composer?
2. Як Composer може бути встановленим в операційній системі?
3. Для якої мови програмування застосовується Composer?
4. Які команди Composer є найвживанішими?
5. Як додати залежність для проєкту за допомогою Composer?

6. Чим відрізняються файли `composer.json` і `composer.lock` та чому останній обов'язково комітиться у Git?
7. Що таке семантичне версіонування (SemVer) і як оператори `^` і `~` обмежують діапазон сумісних версій залежностей?
8. У яких випадках потрібно виконувати `composer dump-autoload` вручну?

Лекція 7. ФРЕЙМВОРК SYMFONY

План лекції:

1. Основні компоненти фреймворка Symfony
2. Структура веб-додатку на Symfony
3. Процес запиту
4. Розгортання проєкту на Symfony
5. Контролери та маршрутизація
6. Doctrine ORM
7. Сервіси та Dependency Injection
8. Тестування у Symfony
9. Видалення пакетів
10. Скидання автозавантаження

Ключові слова: фреймворк, контролер, маршрутизація, структура, тестування.

7.1 Основні компоненти фреймворка Symfony

Symfony – це потужний фреймворк для розробки веб-додатків на мові програмування PHP. Він надає гнучкість та зручність для розробки сучасних веб-додатків, допомагає вирішувати ряд загальних проблем, забезпечує структуру та стандарти для розробки. В цьому розділі ми розглянемо основи фреймворка Symfony, його компоненти та структуру веб-додатку на основі Symfony.

Bundle

Bundle – це основний компонент Symfony, що представляє собою набір пов'язаних файлів (коду, шаблонів, стилів, конфігурації тощо), які разом імплементують певну функціональність. Bundle спрощує розробку, тестування та повторне використання коду, а також допомагає організувати структуру проєкту.

Контролери

Контролери відповідають за обробку запитів від користувача, виконують бізнес-логіку та передають дані до відображення (шаблонів). Вони відіграють

важливу роль у шаблоні проєктування Model-View-Controller (MVC), на якому базується Symfony.

Моделі та сутності

Моделі та сутності представляють бізнес-логіку додатку та відповідають за взаємодію з базою даних. Вони слугують для створення, зберігання та обробки даних, а також їх перетворення у відповідні формати.

Відображення (шаблони)

Відображення, або шаблони, відповідають за відтворення даних, отриманих від контролерів, у візуальному вигляді. Вони можуть використовувати шаблонізатори, такі як Twig, для спрощення та підвищення створення візуального відображення даних.

Маршрутизація

Маршрутизація в Symfony відповідає за визначення шляхів (URL) для доступу до контролерів та передачу запитів від користувачів до відповідних контролерів. Маршрутизація допомагає організувати структуру URL-адрес та забезпечує гнучкість у керуванні доступом до різних частин веб-додатку.

7.2 Структура веб-додатку на Symfony

Каталоги

Основні каталоги в структурі веб-додатку на Symfony:

- ``src``: містить основний код додатку, включаючи контролери, сутності, репозиторії та інші класи;
- ``templates``: містить шаблони відображення, які відповідають за візуальну частину додатку;
- ``config``: містить файли конфігурації додатку, такі як налаштування бази даних, маршрутизації, сервісів тощо;
- ``public``: містить відкриті ресурси, такі як стилі, скрипти, зображення та інші статичні файли;
- ``var``: містить файли журналів, кешу та інші тимчасові файли;

– ``vendor``: містить залежності та бібліотеки сторонніх розробників, які використовуються у проєкті.

7.3 Процес запиту

Коли користувач надсилає запит до веб-додатку на Symfony, відбувається такий процес:

1. запит обробляється фронт-контролером (зазвичай ``public/index.php``);
2. маршрутизація визначає контролер, що відповідає за обробку запиту, на основі url та http-методу;
3. контролер виконує бізнес-логіку, використовуючи моделі та сутності, та отримує дані для відображення;
4. контролер передає отримані дані до відповідного шаблону відображення;
5. шаблон відображення обробляє дані та генерує візуальне подання, яке буде відправлено користувачеві;
6. відповідь з відображенням даних надсилається користувачеві.

7.4 Розгортання проєкту на Symfony

Розгортання проєкту на Symfony включає в себе налаштування середовища, встановлення залежностей та створення базової структури проєкту. В цьому розділі ми розглянемо кроки, необхідні для створення нового проєкту на Symfony з нуля.

Вимоги до середовища

Для розгортання Symfony потрібно мати налаштоване середовище з встановленим PHP (версії 8.2 або вище) та Composer – менеджером залежностей для PHP. Також рекомендується встановити сервер баз даних (наприклад, MySQL або PostgreSQL) та веб-сервер (Apache або Nginx), якщо вони ще не встановлені.

Встановлення Symfony CLI

Встановіть Symfony CLI (командний рядок) за допомогою наступної команди:

```
curl -sS https://get.symfony.com/cli/installer | bash
```

Після встановлення перемістіть виконавчий файл ``symfony`` до каталогу з виконавчими файлами, щоб використовувати його як глобальну команду:

```
mv ~/.symfony/bin/symfony /usr/local/bin/symfony
```

Створення нового проєкту

Створіть новий проєкт Symfony, виконавши наступну команду:

```
symfony new my_project --webapp
```

Замініть ``my_project`` назвою вашого проєкту. Команда створить новий каталог зі структурою Symfony та встановить необхідні залежності.

Запуск веб-сервера

Перейдіть до каталогу нового проєкту:

```
cd my_project
```

Запустіть вбудований веб-сервер Symfony за допомогою команди:

```
symfony server:start
```

Це запустить сервер на порту 8000. Відкрийте веб-браузер та перейдіть за адресою ``http://localhost:8000``, щоб перевірити, що ваш новий проєкт Symfony працює.

7.5 Контролери та маршрутизація

Контролер у Symfony — це звичайний PHP-клас, методи якого обробляють HTTP-запити та повертають об'єкт відповіді. Зв'язок між URL-адресою і контролером описується маршрутом. У Symfony 7 маршрути визначаються прямо в коді контролера через атрибути PHP 8 `#[Route(...)]` — окремий конфігураційний YAML-файл більше не потрібний.

Контролер успадковує `AbstractController` — базовий клас, що надає набір помічників: `$this->json()` для JSON-відповідей, `$this->render()` для Twig-шаблонів, `$this->createNotFoundException()` для відповіді 404 тощо.

Базовий контролер з одним маршрутом

```
// src/Controller/HelloController.php
namespace App\Controller;
```

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Routing\Attribute\Route;
```

```
class HelloController extends AbstractController
{
```

```
#[Route('/api/hello/{name}', methods: ['GET'])]
public function hello(string $name): JsonResponse
{
    return $this->json(['message' => "Hello, $name!"]);
}
}
```

Атрибут `#[Route]` приймає шаблон URL з плейсхолдерами у фігурних дужках та обмеження HTTP-методів через параметр `methods`. Параметри з URL автоматично передаються у метод як аргументи з відповідним типом — Symfony виконує приведення типу за оголошенням `string $name` або `int $id`.

REST-контролер з повним CRUD

Для модуля «Продукти» REST-API стандартно складається з п'яти ендпоінтів: список, перегляд одного, створення, оновлення, видалення. Базовий префікс `/api/products` винесено на рівень класу, окремі методи додають свій суфікс.

```
// src/Controller/ProductsController.php
use Symfony\Component\HttpFoundation\Request;

#[Route('/api/products')]
class ProductsController extends AbstractController
{
    public function __construct(
        private ProductRepository $repository,
        private EntityManagerInterface $em,
    ) {}

    #[Route('', methods: ['GET'])]
    public function findAll(): JsonResponse {
        return $this->json($this->repository->findAll());
    }

    #[Route('/{id}', methods: ['GET'])]
    public function findOne(int $id): JsonResponse {
        $product = $this->repository->find($id);
        if (!$product) {
            throw $this->createNotFoundException("Product $id not found");
        }
        return $this->json($product);
    }

    #[Route('', methods: ['POST'])]
    public function create(Request $request): JsonResponse {
        $data = $request->toArray(); // парсить JSON та валідує його одразу
        $product = new Product($data['title'], $data['price']);
        $this->em->persist($product);
        $this->em->flush();
        return $this->json($product, status: 201);
    }
}
```

```

    }

    #[Route('/{id}', methods: ['DELETE'])]
    public function delete(int $id): JsonResponse {
        $product = $this->repository->find($id);
        $this->em->remove($product);
        $this->em->flush();
        return $this->json(null, status: 204);
    }
}

```

Метод `$this->json()` серіалізує переданий об'єкт у JSON та формує `JsonResponse` зі статусом 200, автоматично встановлює заголовок `Content-Type: application/json` і викликає `Symfony Serializer` під капотом — це звільняє програміста від ручного виклику серіалізатора. Іменований аргумент `status: 201` (PHP 8) перевизначає код стану на «201 Created».

Метод `$request->toArray()` парсить JSON-тіло запиту й одразу перевіряє його коректність, повертаючи асоціативний масив. Це сучасніший спосіб порівняно з `json_decode($request->getContent(), true)`, бо викидає виняток на невалідному JSON замість мовчазного `null`.

Цикл обробки запиту в Symfony

Запит проходить через шари фреймворку у такій послідовності:

1. клієнт надсилає `GET /api/products/42`;
2. `Front-controller public/index.php` приймає запит та створює `HttpKernel`;
3. маршрутизатор зіставляє URL з атрибутом `#[Route]` і знаходить `ProductsController::findOne`;
4. контейнер DI створює контролер і впроваджує залежності (репозиторій, `EntityManager`);
5. `Symfony` викликає метод `findOne(42)`, який повертає `JsonResponse`;
6. `HttpKernel` надсилає відповідь клієнту.

Питання та завдання для самоконтролю до підрозділу:

1. Що таке атрибут `#[Route]` і як він задає URL?
2. Як обмежити маршрут конкретним HTTP-методом?
3. Як отримати параметр шляху у методі контролера?
4. Що повертає метод `$this->json()` і чим відрізняється від `new JsonResponse(...)`?

5. У якій послідовності Symfony обробляє вхідний HTTP-запит?

7.6 Doctrine ORM

Doctrine ORM — найпоширеніший об'єктно-реляційний шар для PHP, інтегрований у Symfony через DoctrineBundle. Працює за патерном Data Mapper: сутність (entity) — звичайний PHP-об'єкт без знання про базу даних, а маппер (EntityManager) керує збереженням і завантаженням об'єктів. Це відрізняє Doctrine від Active Record (Eloquent у Laravel), де об'єкт сам містить логіку збереження.

Конфігурація підключення до БД задається у файлі `env` через змінну середовища `DATABASE_URL` — наприклад, `mysql://user:pass@127.0.0.1:3306/mydb`. Зберігання DSN у `env` (а не в коді) відповідає принципу 12-Factor App.

Сутність (Entity)

Сутність — клас з атрибутами Doctrine, що описують відображення на колонки таблиці БД. Атрибут `#[ORM\Entity]` позначає клас як керовану сутність; `#[ORM\Column]` — окремі поля; `#[ORM\Id]` + `#[ORM\GeneratedValue]` — первинний ключ з автоінкрементом.

```
// src/Entity/Product.php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: ProductRepository::class)]
#[ORM\Table(name: 'products')]
class Product
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private string $title;

    #[ORM\Column(type: 'decimal', precision: 10, scale: 2)]
    private string $price;

    #[ORM\Column]
    private bool $inStock = true;

    #[ORM\Column(type: 'datetime_immutable')]
```

```

private \DateTimeImmutable $createdAt;

public function __construct(string $title, string $price) {
    $this->title = $title;
    $this->price = $price;
    $this->createdAt = new \DateTimeImmutable();
}

public function getId(): ?int { return $this->id; }
public function getTitle(): string { return $this->title; }
public function getPrice(): string { return $this->price; }
public function setPrice(string $price): void { $this->price =
$price; }
}

```

Властивості сутності оголошені `private` з типами PHP 8 (`?int`, `string`, `bool`, `\DateTimeImmutable`). Геттери і сеттери відкривають доступ ззовні — це класичний PHP-патерн інкапсуляції.

Зверніть увагу: ціна `$price` оголошена як `string`, хоча в базі це `decimal(10,2)`. Це стандартна практика для грошових сум — щоб уникнути втрати точності при представленні дробових чисел типом `float`. PHP, як і більшість мов, зберігає `float` як IEEE 754, який не може точно представити 0.1, 0.2 та інші типові дробові ціни ($0.1 + 0.2 \neq 0.3$ у `float`-арифметиці). Doctrine при INSERT/UPDATE передає рядок «24990.50» у БД без перетворення; при SELECT повертає рядок назад. Арифметика з грошима в PHP робиться через розширення `bcmath`: `bcadd($a, $b, 2)`, `bcmul($a, $b, 2)`.

Репозиторій (Repository)

Репозиторій — клас із методами пошуку для конкретної сутності. Базові методи `find($id)`, `findAll()`, `findBy([...])`, `findOneBy([...])` надає клас-предок `ServiceEntityRepository`. Власні методи запитів реалізуються через Query Builder або мову запитів DQL (Doctrine Query Language).

```

// src/Repository/ProductRepository.php
namespace App\Repository;

use App\Entity\Product;
use
Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

class ProductRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry) {
        parent::__construct($registry, Product::class);
    }
}

```

```

    public function findInStock(): array {
        return $this->createQueryBuilder('p')
            ->andWhere('p.inStock = true')
            ->orderBy('p.createdAt', 'DESC')
            ->getQuery()
            ->getResult();
    }
}

```

Міграції — автоматично з diff схеми

Doctrine Migrations порівнює поточну схему БД з описаними сутностями та генерує SQL-міграцію автоматично. Кожна міграція — клас з методами `up()` (впровадити зміни) та `down()` (скасувати).

Команди міграцій:

- Створити нову міграцію з diff поточних сутностей та БД

```
php bin/console make:migration
```

- Застосувати всі непримінені міграції

```
php bin/console doctrine:migrations:migrate
```

- Скасувати останню міграцію

```
php bin/console doctrine:migrations:migrate prev
```

- Створити порожню БД (виконується одного разу)

```
php bin/console doctrine:database:create
```

CRUD-операції через EntityManager

EntityManager — головний об'єкт Doctrine, що керує життєвим циклом сутностей. Зміни накопичуються у пам'яті до виклику `flush()` — лише тоді Doctrine виконує реальні SQL-запити в одній транзакції. Це дає вигоду у продуктивності та консистентності.

```

// Створення
$product = new Product('Ноутбук', '24990.50');
$entityManager->persist($product); // запам'ятати намір зберегти
$entityManager->flush();             // виконати INSERT

// Читання
$product = $repository->find(42);
$inStock = $repository->findInStock();

// Оновлення
$product->setPrice('22990.00');
$entityManager->flush(); // persist не потрібен — об'єкт вже
керований

```

```
// Видалення
$entityManager->remove($product);
$entityManager->flush();
```

Питання та завдання для самоконтролю до підрозділу:

1. Що таке сутність у Doctrine та як її описати атрибутами PHP 8?
2. Як створити та застосувати міграцію?
3. Чим відрізняються методи `persist()` та `flush()`?
4. Як написати свій метод пошуку у репозиторії за допомогою Query Builder?
5. Чим Doctrine ORM відрізняється від Active Record (Eloquent у Laravel)?

7.7 Сервіси та Dependency Injection

Сервіс — будь-який PHP-об'єкт, що виконує конкретну роль у застосунку: робота з БД, надсилання email, логування, обробка платежів. У Symfony сервіси живуть у контейнері впровадження залежностей (Service Container, DIC), який автоматично створює та постачає їх там, де вони потрібні.

Принцип Dependency Injection (DI) полягає в тому, що клас НЕ створює свої залежності сам, а отримує їх ззовні через конструктор. Це дає слабе зчеплення (loose coupling), легке тестування (можна підміняти залежність на мок) та конфігурованість (одну залежність можна замінити іншою без зміни коду споживача).

Без DI vs з DI

```
// БЕЗ DI — жорстке зчеплення, тяжко тестувати
class ProductService {
    private ProductRepository $repo;
    public function __construct() {
        $this->repo = new ProductRepository(); // зашите
        всередину
    }
}

// З DI — слабе зчеплення, легко мокати у тестах
class ProductService {
    public function __construct(private ProductRepository $repo)
    {}
}
```

Autowiring у Symfony

Symfony 7 за замовчуванням увімкнений autowiring: контейнер аналізує type-hints конструктора та автоматично постачає правильні залежності. Жодних XML/YAML-конфігурацій сервісів писати не потрібно — фреймворк сам знаходить класи у src/, реєструє їх як сервіси та зв'язує між собою.

```
# config/services.yaml — це все, що потрібно
services:
    _defaults:
        autowire: true           # автоматично визначати залежності за
типами
        autoconfigure: true     # автоматично реєструвати тегування
    App\:
        resource: '../src/'
```

Архітектурний шаблон: Controller → Service → Repository → Database

Класична багат шарова архітектура виносить бізнес-логіку у сервіси, залишаючи контролери тонкими (приймають запит, делегують сервісу, повертають відповідь). Така структура полегшує юніт-тестування та повторне використання логіки в інших точках (CLI-команди, фонові завдання).

```
// src/Service/ProductService.php
namespace App\Service;

use App\Entity\Product;
use App\Repository\ProductRepository;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\Validator\Validator\ValidatorInterface;

class ProductService
{
    public function __construct(
        private ProductRepository $repository,
        private EntityManagerInterface $em,
        private ValidatorInterface $validator,
    ) {}

    public function createProduct(array $data): Product {
        $product = new Product($data['title'], $data['price']);
        $errors = $this->validator->validate($product);
        if (count($errors) > 0) {
            throw new ValidationFailedException($product,
$errors);
        }
        $this->em->persist($product);
        $this->em->flush();
        return $product;
    }
}
```

```
// src/Controller/ProductsController.php
class ProductsController extends AbstractController {
    public function __construct(private ProductService $service)
    {}

    #[Route('', methods: ['POST'])]
    public function create(Request $request): JsonResponse {
        $data = $request->toArray();
        $product = $this->service->createProduct($data);
        return $this->json($product, status: 201);
    }
}
```

Контролер тут — тонка адаптерна обгортка: десеріалізує JSON, делегує бізнес-логіку сервісу, формує HTTP-відповідь. У юніт-тестах ProductService отримує замоковані залежності замість реальних БД та валідатора.

Питання та завдання для самоконтролю до підрозділу:

1. Що таке Dependency Injection і навіщо він потрібен?
2. Як працює autowiring у Symfony?
3. Які переваги дає шар Service між контролером і репозиторієм?
4. Як замикає залежність ProductRepository у юніт-тесті сервісу?

7.8 Валідація та серіалізація

Symfony Validator перевіряє об'єкти на відповідність обмеженням (constraints), описаним атрибутами безпосередньо на властивостях класу. Це декларативний підхід — правила валідації живуть поряд з полями, а не в окремих процедурних перевірках.

Атрибути валідації

```
// src/Entity/Product.php
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

class Product {
    #[ORM\Column(length: 255)]
    #[Assert\NotBlank]
    #[Assert\Length(min: 2, max: 255)]
    private string $title;

    #[ORM\Column(type: 'decimal', precision: 10, scale: 2)]
    #[Assert\Positive(message: 'Ціна має бути більшою за нуль')]
    private string $price;

    #[ORM\Column(length: 180, nullable: true)]
    #[Assert\Email]
```

```
private ?string $contactEmail = null;
}
```

Обмеження `#[Assert\NotBlank]` забороняє порожнє значення; `#[Assert\Length]` обмежує довжину рядка; `#[Assert\Positive]` вимагає додатнє число; `#[Assert\Email]` перевіряє формат email-адреси. Параметр `message` у будь-якому атрибуті задає текст помилки, що повертається користувачеві.

Виклик валідації

У сервісі валідація завершується викиданням винятку — формування HTTP-відповіді не входить до зони відповідальності бізнес-логіки:

```
use Symfony\Component\Validator\Validator\ValidatorInterface;
use
Symfony\Component\Validator\Exception\ValidationFailedException;

$errors = $validator->validate($product);
if (count($errors) > 0) {
    throw new ValidationFailedException($product, $errors);
}
```

У контролері (якщо НЕ застосовано атрибут `#[MapRequestPayload]`, див. наступний підрозділ) перевірка та відповідь з кодом 422 формується явно:

```
$errors = $validator->validate($product);
if (count($errors) > 0) {
    return $this->json($errors, status: 422);
}
```

Symfony автоматично серіалізує об'єкт `ConstraintViolationList` у JSON-масив через вбудований нормалізатор — окремо формувати масив повідомлень не потрібно. Стандартний HTTP-код для помилок валідації — 422 `Unprocessable Entity` (а не загальний 400 `Bad Request`, який доречний радше для синтаксично невалідного запиту).

Symfony Serializer

Serializer перетворює PHP-об'єкти у JSON / XML / CSV та назад. Метод `$this->json()` у контролері використовує Serializer під капотом, але його можна викликати напряму — наприклад, для десеріалізації тіла запиту у сутність.

```
use Symfony\Component\Serializer\SerializerInterface;

// Об'єкт → JSON
$json = $serializer->serialize($product, 'json');

// JSON → об'єкт сутності
$product = $serializer->deserialize(
    $request->getContent(),
    Product::class,
```

```
'json'
);
```

Сучасний підхід — атрибут #[MapRequestPayload]

Symfony 7 пропонує атрибут #[MapRequestPayload], який об'єднує всі чотири кроки (десеріалізація, валідація, обробка помилок, type-hint) в одне оголошення параметра методу. Контролер стає однорядковим:

```
use Symfony\Component\HttpKernel\Attribute\MapRequestPayload;

#[Route('', methods: ['POST'])]
public function create(
    #[MapRequestPayload] Product $product,
    EntityManagerInterface $em,
): JsonResponse {
    $em->persist($product);
    $em->flush();
    return $this->json($product, status: 201);
}
```

Що робить #[MapRequestPayload]: десеріалізує JSON-тіло запиту у Product, прогоняє його через ValidatorInterface, і у разі невалідних даних автоматично повертає відповідь 422 Unprocessable Entity з масивом помилок — без жодного рядка коду в контролері. Це стандартний рекомендований спосіб для нових проєктів на Symfony 7.

Явний цикл (для розуміння внутрішньої механіки)

Атрибут #[MapRequestPayload] зручний у роботі, але корисно знати, з чого він складається. Розгорнутий варіант контролера явно прописує всі чотири кроки:

```
#[Route('', methods: ['POST'])]
public function create(
    Request $request,
    SerializerInterface $serializer,
    ValidatorInterface $validator,
    EntityManagerInterface $em,
): JsonResponse {
    // 1. Десеріалізація JSON у сутність
    $product = $serializer->deserialize(
        $request->getContent(),
        Product::class,
        'json'
    );

    // 2. Валідація — 422 із масивом помилок при невдачі
    $errors = $validator->validate($product);
    if (count($errors) > 0) {
        return $this->json($errors, status: 422);
    }
}
```

```
// 3. Збереження у БД
$em->persist($product);
$em->flush();

// 4. Відповідь зі створеною сутністю
return $this->json($product, status: 201);
}
```

Питання та завдання для самоконтролю до підрозділу:

1. Які атрибути валідації ви знаєте і за що відповідає кожен?
2. Що повертає метод `$validator->validate()` і як інтерпретувати результат?
3. Як перетворити JSON-тіло запиту у PHP-об'єкт через `Serializer`?
4. Чим декларативна валідація через атрибути зручніша за процедурну?
5. Що робить атрибут `#[MapRequestPayload]` і чому його використання — стандарт для нових Symfony 7 проєктів?
6. Який HTTP-статус використовується для помилок валідації?

7.9 Тестування у Symfony

Тестування — невід'ємна частина професійної розробки на Symfony. Воно поділяється на три рівні: `unit`-тести перевіряють окремі класи у ізоляції; `integration`-тести — взаємодію кількох компонентів (наприклад, сервіс із підключенням до БД); `functional`-тести — повний HTTP-стек (контролери, маршрути, серіалізація, валідація) від запиту до відповіді. Усі три рівні запускаються одним інструментом — `PHPUnit`.

Symfony надає базові класи `KernelTestCase` (доступ до сервіс-контейнера) та `WebTestCase` (емуляція HTTP-клієнта без реального веб-сервера). Тести розташовуються у каталозі `tests/`, дзеркально структурі `src/`. Команда запуску — `php bin/phpunit` — автоматично знаходить і виконує усі тестові класи. Тестове середовище Symfony активується змінною `APP_ENV=test` із окремим файлом `env.test` (тестова БД, мокнуті зовнішні API, відключений email-mailer).

Unit-тест на сервіс створює екземпляр класу, передає мокнуті залежності, викликає метод, перевіряє результат через `$this->assertSame(...)` чи `$this->assertTrue(...)`. Жодних звернень до БД, файлової системи чи мережі — це гарантує швидкість (мілісекунди на тест) і незалежність від оточення. Для

ізоляції зовнішніх викликів застосовують моки PHPUnit — `createMock(ClassName::class)` — або власні `test doubles`.

Functional-тест перевіряє цілий маршрут із реальною БД (зазвичай SQLite в пам'яті для швидкості). У `WebTestCase` створюється тестовий клієнт `$client = static::createClient()`, що надсилає HTTP-запити безпосередньо до Symfony-ядра без TCP-сокетів. Приклад: `$client->request('GET', '/api/products/1');` `$this->assertResponseIsSuccessful();` `$this->assertJsonContains(['name' => 'Notebook']);`. Аспекти `assertResponseStatusCodeSame(404)`, `assertJsonContains(...)`, `assertResponseHasHeader(...)` є частиною пакета `symfony/test-pack`. Для очищення стану між тестами використовують `DAMA\DoctrineTestBundle`, що запускає кожен тест у транзакції та робить `rollback` наприкінці — швидше за перестворення схеми.

Тестові фікстури (`doctrine/data-fixtures-bundle`) дозволяють завантажувати передзавантажені дані БД для повторюваних сценаріїв через `php bin/console doctrine:fixtures:load --env=test`. Покриття коду (`code coverage`) генерується через `php bin/phpunit --coverage-html coverage/` (потрібне розширення `Xdebug` або `PCOV`). Промислова практика — мінімум 70–80% покриття для бізнес-логіки сервісів; контролери та інфраструктурний код покриваються functional-тестами. 1 У 1 рекомендовано написати щонайменше 5 unit-тестів на сервіси та 3 functional-тести на ключові API-ендпоїнти.

Порівняння Symfony та NestJS

Нижче — попередній огляд відповідностей між Symfony 7 та NestJS 10. Детальне порівняння — у підрозділах 8.11 (Робота з базою даних: Prisma ORM) та 8.12 (Валідація та DTO).

7.10 Питання та завдання для самоконтролю

1. Яка структура каталогів Symfony-проекту і за що відповідає кожен з них (`src`, `config`, `public`, `var`, `vendor`)?

2. Що таке front-controller у Symfony і чому всі запити проходять через `public/index.php`?
3. Як визначити маршрут до контролера за допомогою атрибута `#[Route]`?
4. Як обмежити маршрут конкретним HTTP-методом і як отримати параметри з URL?
5. Що повертає метод `$this->json()` контролера і чим відрізняється від `new JsonResponse(...)`?
6. Чим відрізняється сутність і репозиторій у Doctrine? Що таке `ServiceEntityRepository`?
7. Що таке `EntityManager` і як він керує життєвим циклом сутності?
8. Чому потрібно викликати `flush()` після `persist()`? Як це пов'язано з транзакціями?
9. Як створити та застосувати міграцію з поточних сутностей?
10. Чим Doctrine ORM (Data Mapper) відрізняється від Active Record (Eloquent у Laravel)?
11. У чому полягає принцип `Dependency Injection`?
12. Що таке `autowiring` і як він спрощує реєстрацію сервісів у Symfony?
13. Які атрибути валідації використовуються для перевірки рядків та чисел?
14. Як перевірити вхідні дані запиту через `ValidatorInterface`?
15. Як перетворити JSON-тіло запиту у PHP-об'єкт через `Serializer`?
16. Яку роль виконує шар `Service` між контролером і репозиторієм?
17. Як Symfony Validator порівнюється з `class-validator` у NestJS?
18. Як Doctrine ORM порівнюється з Prisma за способом опису сутностей?

Лекція 8. СЕРВЕРНИЙ JAVASCRIPT: NODE.JS

План лекції:

1. Вступ
2. Блокуючі введення/виведення та проблема C10K
3. Node.js: платформа та цикл подій
4. CPU-ємні завдання та воркери
5. Директиви Import та Export
6. Cookie та сесії в Node.js
7. npm та package.json
8. TypeScript: основи для серверної розробки
9. NestJS та Prisma ORM
10. Створення проєкту та структура NestJS
11. REST API у NestJS: контролери та сервіси
12. Робота з базою даних: Prisma ORM
13. Валідація та DTO у NestJS
14. Гварди, перехоплювачі та JWT-аутентифікація

***Ключові слова:** фреймворк, javascript, воркери, структура, база даних, гварди, переховлювачі.*

8.1 Вступ

Веб-застосунки, написані за клієнтом-серверною архітектурою, працюють за наступною схемою. Клієнт запитує потрібний ресурс у сервера і сервер надсилає ресурс у відповідь. У цій схемі сервер, відповівши на запит, перериває з'єднання.

Така модель ефективна, оскільки кожен запит до сервера споживає ресурси (пам'ять, процесорний час і т.д.). Щоб обробляти кожен наступний запит від клієнта, сервер повинен завершити обробку попереднього.

Чи означає це, що сервер може обробляти лише один запит? Ні, це не так! Коли сервер отримує новий запит, він створює окремий **потік** для його обробки.

Потік – це час і ресурси, які CPU виділяє для виконання невеликого блоку інструкцій. З урахуванням сказаного сервер може обробляти кілька запитів одночасно, але тільки по одному на потік (рис. 8.1). Така модель також називається **thread-per-request model**.

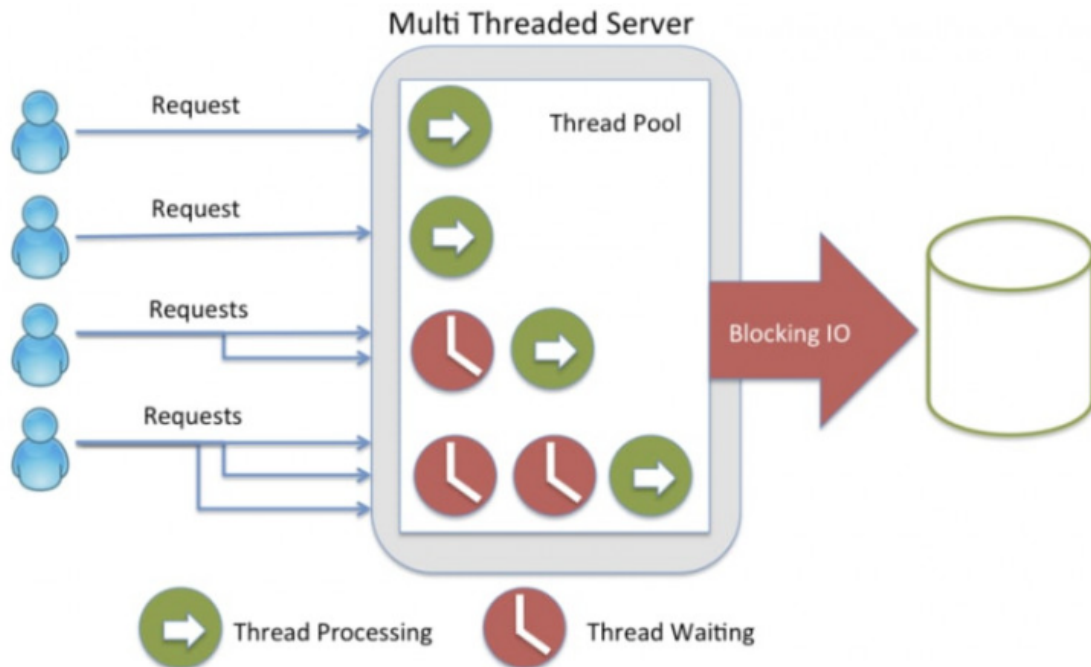


Рисунок 8.1 – Діаграма потоків обробки запитів сервером

Для обробки N запитів серверу потрібне N потоків. Якщо сервер отримує $N+1$ запитів, тоді він повинен чекати доки один із потоків не стане доступним.

На малюнку вище, сервер може обробляти до 4 запитів (потоків) одночасно і коли він отримує наступні 3 запити, ці запити повинні чекати, доки будь-який з цих 4 потоків не стане доступним.

Один із способів позбутися обмежень – додати більше ресурсів (пам'яті, ядер процесора і тощо) на сервер, але це не єдиний спосіб. І, звичайно, не забуваємо про технологічні обмеження.

8.2 Блокуючі введення/виведення та проблема C10K

Обмежена кількість потоків на сервері не єдина проблема. Можливо, вам стало цікаво, чому один потік не може обробляти кілька запитів одночасно? Все через **блокуючі операції введення/виведення**.

Припустимо, ви розробляєте онлайн-магазин і вам потрібна сторінка, де користувач може переглядати список усіх товарів.

Користувач робить запит на <http://yourstore.com/products> і сервер формує HTML файл з усіма продуктами з бази даних у відповідь. Зовсім не складно, так?

– Коли користувач робить запит на /products особливий метод або функція повинна виконатися, щоб обробити запит. Маленький фрагмент коду (ваш або фреймворка) аналізує URL-адресу запиту та шукає відповідний метод чи функцію. **Потік працює.**

– Тепер потрібний метод чи функція виконуватимуться. **Потік працює.**

– Оскільки ви хороший розробник, ви зберігаєте всі системні логи у файл, ну і звичайно ж, щоб бути впевненими, що роутер виконує потрібний метод/функцію - ви також записуєте в лог рядок "Method X executing!!». Але все це блокуючі операції введення/виводу **Потік чекає.**

– Усі логи збережені та наступні рядки функції виконуються. **Потік працює знову.**

– Час звертатися до бази даних і отримувати всі продукти - простий запит, начебто SELECT * FROM products виконує свою роботу, але вгадайте що? Так-так, це блокуюча операція введення/виводу. **Потік чекає.**

– Ви отримали масив або список всіх продуктів, але переконайтеся, що ви все це записали в лог. **Потік чекає.**

– Тепер у вас є всі продукти і настав час рендерувати шаблон для майбутньої сторінки, але перед цим вам потрібно їх прочитати. **Потік чекає.**

– Шаблонізатор рендерингу робить свою роботу і надсилає відповідь клієнту. **Потік працює знову.**

Наскільки повільні операції введення/виводу? Це залежить від конкретної операції (табл. 8.1).

Таблиця 8.1 – Ресурси процесора для блокуючих операцій

Операція	Кількість тактів CPU
CPU Registers	3
L1 Cache	8
L2 Cache	12
RAM	150
Disk	30 000 000
Network	250 000 000

Операції мережі та читання з диска надто повільні. Уявіть, скільки запитів або звернень до зовнішніх API ваша система могла б обробити за цей час.

Підбиваючи підсумки: операції введення/виводу змушують потік чекати й витрачати ресурси марно.

C10k (англ. *C10k*; *10k connections* – проблема 10 тисяч з'єднань)

У ранні 2000-і серверні та клієнтські машини були повільними. Проблема виникала при паралельній обробці 10 000 клієнтських з'єднань до однієї машини.

Але чому традиційна модель *thread-per-request* (потік на запит) не могла вирішити цієї проблеми? Що ж, давайте використовуємо трохи математики.

Нативна реалізація потоків виділяє більше 1 Мб пам'яті на потік, виходячи з цього – для 10 тисяч потоків потрібно 10 Гб оперативної пам'яті, і це лише для стека потоків. Так, і не забувайте, ми на початку 2000-х!

У наші дні серверні та клієнтські комп'ютери працюють швидше та ефективніше і майже будь-яка мова програмування або фреймворк справляються з цією проблемою. Але фактично проблема не вичерпана. Для 10 мільйонів клієнтських з'єднань до однієї машини проблема повертається знову (але тепер [C10M Problem](#)).

8.3 Node.js: платформа та цикл подій

Node.js дійсно вирішує проблему C10K... але як?!

Серверний JavaScript не був чимось новим і незвичайним на початку 2000-х, на той момент вже існували продажі поверх JVM (Java Virtual Machine) – RingoJS та AppEngineJS, що працювали на моделі *thread-per-request*.

Але якщо вони не змогли вирішити проблему, тоді як Node.js зміг? Все тому, що JavaScript **однопоточний**.

Node.js — це серверна платформа, що працює на механізмі Google Chrome-V8, який здатен компілювати Код JavaScript в машинний код.

Node.js використовує модель, орієнтовану на події, і архітектуру **неблокуючого введення/виведення**, що робить його стрімким і ефективним. Це

не фреймворк і не бібліотека, це середовище виконання JavaScript, програмна платформа.

Давайте напишемо маленький приклад:

```
// Importing native http module
const http = require('http');

// Creating a server instance where every call
// the message 'Hello World' is responded to the client
const server = http.createServer(function(request, response) {
  response.write('Hello World');
  response.end();
});

// Listening port 8080
server.listen(8080);
```

Non-blocking I/O

Node.js використовує операції, що не блокують введення/виведення, що ж це означає:

- Головний потік не блокуватиметься операціями введення/виводу.
- Сервер продовжуватиме обслуговувати запити.
- Нам доведеться працювати з **асинхронним кодом**.

Давайте напишемо приклад, у якому на запит до /homесервер у відповідь надсилає HTML сторінку, а для всіх інших запитів - 'Hello World'. Щоб відіслати HTML сторінку спочатку її потрібно прочитати з файлу.

home.html

```
<html>
<body>
<h1>This is home page</h1>
</body>
</html>
```

index.js

```
const http = require('http');
const fs = require('fs');

const server = http.createServer(function(request, response) {
  if (request.url === '/home') {
    fs.readFile(`${__dirname}/home.html`, function (err, content) {
      if (!err) {
        response.setHeader('Content-Type', 'text/html');
        response.write(content);
      } else {
        response.statusCode = 500;
        response.write('An error has occurred');
      }
    });
  }
});
```

```

}

response.end();
});
} else {
response.write('Hello World');
response.end();
}
});

server.listen(8080);

```

Якщо запитувана url-адреса /home, тоді використовується нативний модуль fs для читання файлу home.html.

Функції, що потрапляють у http.createServer і fs.readFile як аргументи – **колбеки** (callbacks). Ці функції будуть виконані в одному з моментів у майбутньому (Перша, як тільки сервер отримає запит, а друга — коли файл буде прочитаний з диска і поміщений у буфер).

Поки файл зчитується з диска, Node.js може обробляти інші запити і навіть зчитувати файл знову і все це в одному потоці. Давайте розберемося, як це працює.

Цикл подій – це буквально нескінченний цикл всередині Node.js, і насправді один потік. Фрагмент циклу подій Node.js наведений на рис. 8.2.

Фрагмент циклу подій Node.js

Один потік JavaScript делегує I/O в libuv, а callbacks повертаються в черги подій

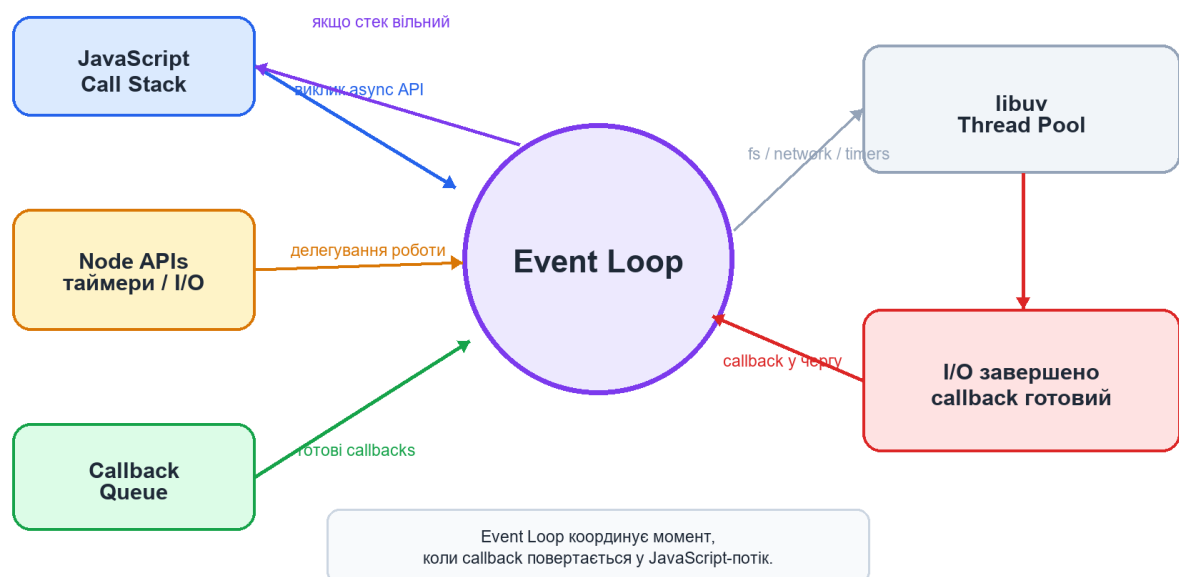


Рисунок 8.2 – Фрагмент циклу подій Node.js

Libuv – C бібліотека, яка реалізує цей патерн і є частиною ядра Node.js.

Цикл подій має 6 фаз, кожне виконання всіх 6 фаз називають **tick**-ом (рис. 8.3).

Фази циклу подій Node.js

Один tick проходить послідовні фази; між фазами виконуються microtasks

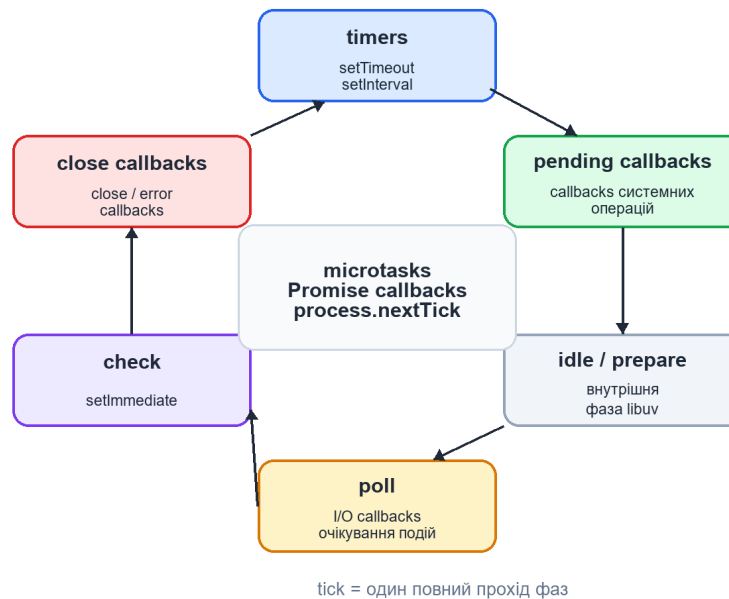


Рисунок 8.3 – Фази циклу подій

- **timers**: у цій фазі виконуються callback-функції, заплановані методами `setTimeout()` та `setInterval()`;
- **pending callbacks**: виконуються майже всі callback-функції, за винятком подій `close`, таймерів та `setImmediate()`;
- **idle, prepare**: використовується лише для внутрішніх цілей;
- **poll**: відповідальний за отримання нових подій вводу/виводу. Node.js може блокуватись на цьому етапі;
- **check**: callback-функції, викликані методом `setImmediate()`, виконуються на цьому етапі;
- **close callbacks** : наприклад, `socket.on('close',...)`;

Зверніть увагу :

Коли циклу подій потрібно виконати операцію введення/виводу він використовує потік ОС з пулу потоку (thread pool), а коли завдання виконано, callback ставиться в чергу під час фази *pending callbacks* (рис. 8.4).

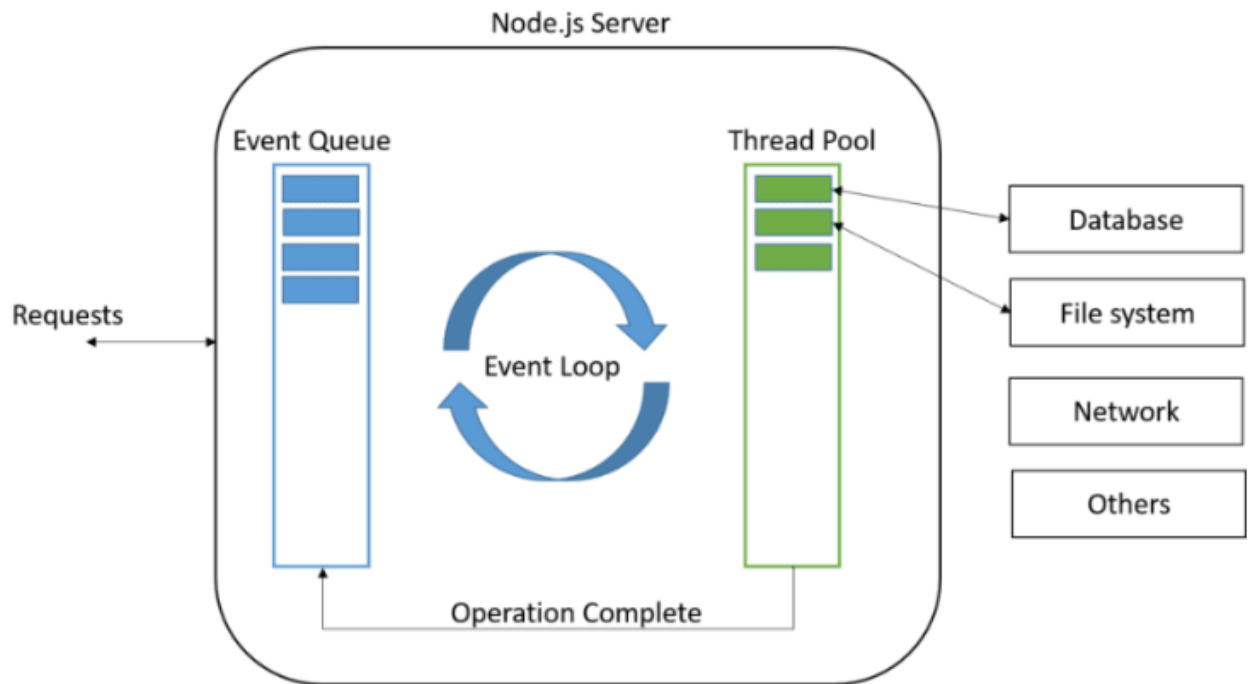


Рисунок 8.4 – Черга callback пулу потоку

8.4 CPU-ємні завдання та воркери

Розглянемо приклад API для обчислення простих чисел — задачу, що інтенсивно використовує процесор:

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Рисунок 8.5 – Матриця простих чисел

Дано число N , API повинен обчислювати та повертати перші N простих чисел у список (або масив).

primes.js

```
function isPrime(n) {
  for(let i = 2, s = Math.sqrt(n); i <= s; i++) {
    if(n % i === 0) return false;
  }
  return n > 1;
}

function nthPrime(n) {
  let counter = n;
  let iterator = 2;
  let result = [];

  while(counter > 0) {
    isPrime(iterator) && result.push(iterator) && counter--;
    iterator++;
  }

  return result;
}

module.exports = { isPrime, nthPrime };
```

index.js

```
const http = require('http');
const url = require('url');
const primes = require('./primes');

const server = http.createServer(function (request, response) {
  const { pathname, query } = url.parse(request.url, true);

  if (pathname === '/primes') {
    const result = primes.nthPrime(query.n || 0);
    response.setHeader('Content-Type', 'application/json');
    response.write(JSON.stringify(result));
    response.end();
  } else {
    response.statusCode = 404;
    response.write('Not Found');
    response.end();
  }
});

server.listen(8080);
```

Файл `index.js` відповідає за створення сервера і використовує модуль `prime.js` для обробки кожного запиту на `/primes`. Число `N` прокидається через рядок запиту в URL-адресі.

Щоб отримати перші 20 простих чисел нам потрібно зробити запит на `http://localhost:8080/primes?n=20`.

Припустимо, до нас стукають 3 клієнти і намагаються отримати доступ до нашого API, що не блокується введенням/виводом (див. рис. 8.6):

- перший запитує 5 простих чисел кожну секунду;
- другий запитує 1000 простих чисел кожну секунду;
- третій запитує 10 000 000 000 простих чисел, але...

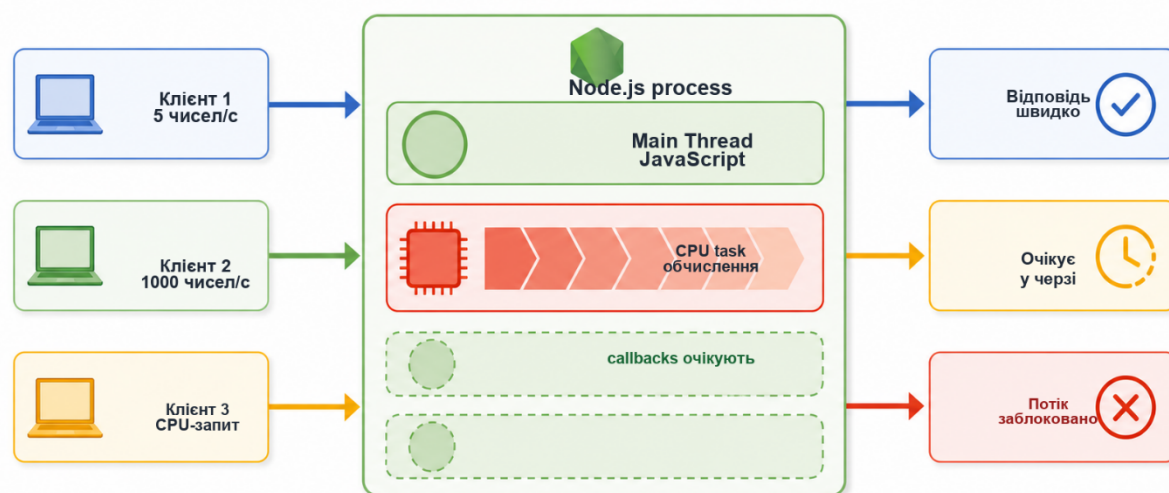


Рисунок 8.6 – Блокування головного потоку

Коли третій клієнт надсилає запит – головний потік блокується і це головна ознака проблеми **CPU-ємних завдань**. Коли головний потік зайнятий виконанням важкої задачі він стає недоступним для інших завдань.

Але як щодо `libuv`? Якщо ви пам'ятаєте, ця бібліотека допомагає Node.js виконувати операції введення/виводу за допомогою потоків ОС уникаючи блокування головного потоку і ви абсолютно праві, це вирішення нашої проблеми, але для того, щоб це стало можливим, наш модуль повинен бути написаний мовою C++, щоб `libuv` могла з ним працювати.

На щастя, починаючи з v10.5 до Node.js доданий вбудований модуль **Worker Threads**.

Воркери корисні для виконання CPU-ємних JavaScript операцій; не використовуйте їх для операцій вводу/виводу, вже вбудовані в Node.js механізми більш ефективно справляються з такими завданнями, ніж Worker thread.

Перепишемо приклад з використанням Worker Threads. Файл `primes-workerthreads.js` — копія `primes.js`, обгорнута у worker-протокол: `primes-workerthreads.js`

```
const { workerData, parentPort } = require('worker_threads');

function isPrime(n) {
  for(let i = 2, s = Math.sqrt(n); i <= s; i++)
    if(n % i === 0) return false;
  return n > 1;
}

function nthPrime(n) {
  let counter = n;
  let iterator = 2;
  let result = [];

  while(counter > 0) {
    isPrime(iterator) && result.push(iterator) && counter--;
    iterator++;
  }

  return result;
}

parentPort.postMessage(nthPrime(workerData.n));
```

`index-workerthreads.js`

```
const http = require('http');
const url = require('url');
const { Worker } = require('worker_threads');

const server = http.createServer(function (request, response) {
  const { pathname, query } = url.parse(request.url, true);

  if (pathname === '/primes') {
    const worker = new Worker('./primes-workerthreads.js', {
      workerData: { n: query.n || 0 } });

    worker.on('error', function () {
      response.statusCode = 500;
      response.write('Oops there was an error...');
      response.end();
    });

    let result;
    worker.on('message', function (message) {
      result = message;
    });

    worker.on('exit', function () {
      response.setHeader('Content-Type', 'application/json');
      response.write(JSON.stringify(result));
    });
  }
});
```

```

    response.end();
  });
} else {
  response.statusCode = 404;
  response.write('Not Found');
  response.end();
}
});

server.listen(8080);

```

У файлі `index-workerthreads.js` при кожному запиті `/primes` створюється екземпляр класу `Worker` (з нативного модуля `worker_threads`) для вивантаження і виконання файлу `primes-workerthreads.js` в потік воркера. Коли список простих чисел прорахований і готовий, ініціюється подія `message` – результат потрапляє в головний потік через те, що у воркера не залишилося роботи, він також ініціює подію `exit`, дозволяючи основному потоку надсилати дані клієнту.

Тепер давайте випробуємо наш приклад знову і подивимося, що станеться (див. рис. 8.7).

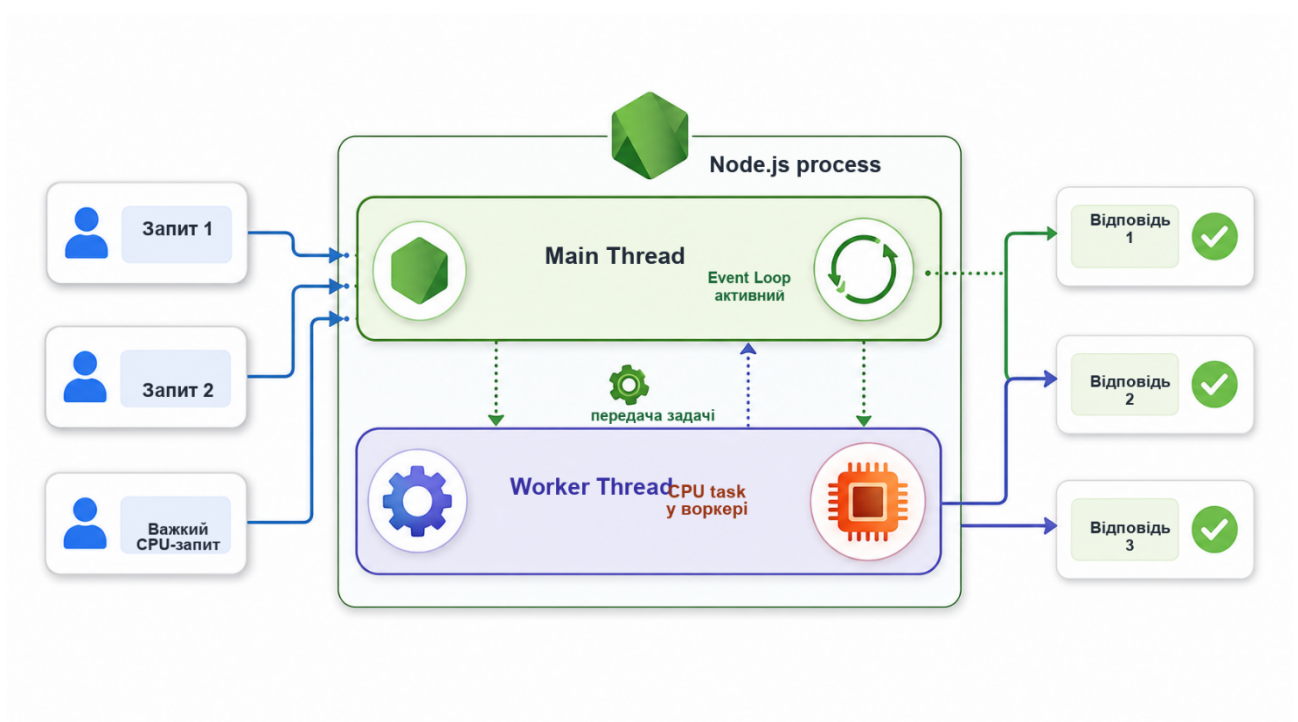


Рисунок 8.7 – Робота основного потоку

Основний потік більше не блокується

Тепер все працює як треба, але зловживати воркерами без жодних на те причин все ж таки не найкраща практика, створювати потоки не дешево задоволення. Обов'язково створіть пул потоків перед цим.

8.5 Директиви Import та Export

Import

Зазвичай ми маємо список того, що хочемо імпортувати, у фігурних дужках **import** {...}, наприклад:

```
import {sayHi, sayBye} from './say.js';  
sayHi('John'); // Hello, John!  
sayBye('John'); // Bye, John!
```

Але якщо імпортувати потрібно багато чого, ми можемо імпортувати все відразу як об'єкта, використовуючи **import * as <obj>**. Наприклад:

```
import * as say from './say.js';  
say.sayHi('John');  
say.sayBye('John');
```

На перший погляд, «імпортувати все» виглядає дуже зручно, не треба писати зайвого, навіть нам взагалі може знадобитися явно перераховувати список того, що потрібно імпортувати?

На це є кілька причин.

1. Сучасні інструменти складання (webpack та інші) збирають модулі разом і оптимізують їх, прискорюючи завантаження і видаляючи код, що не використовується.

Припустимо, ми додали до нашого проєкту сторонню бібліотеку say.js з безліччю функцій:

```
export function sayHi() {...}  
export function sayBye() {...}  
export function becomeSilent() {...}
```

Тепер, якщо з цієї бібліотеки у проєкті ми використовуємо лише одну функцію:

```
import {sayHi} from './say.js';
```

Тоді оптимізатор побачить, що інші функції не використовуються, і видалить інші із зібраного коду, тим самим роблячи код меншим. Це називається "tree-shaking".

2. Очевидно, перераховуючи те, що хочемо імпортувати, ми отримуємо більш короткі імена функцій: sayHi() замість say.sayHi().
3. Явний перелік імпортів робить код зрозумілішим, дозволяє побачити, що саме і де використовується. Це спрощує підтримку та рефакторинг коду.

Export

Ми можемо позначити будь-яке оголошення як експортоване, розмістивши `export` перед ним, будь то змінна, функція або клас.

Наприклад, всі наступні екпорти допустимі:

```
// експорт масиву
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec'];

// експорт константи
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// експорт класу
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

Експорт окремо від оголошення.

Можна також написати `export` окремо. Тут ми спочатку оголошуємо, а потім екпортуємо:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye};
```

Або, технічно, ми також можемо розмістити `export` вище функції.

8.6 Cookie та сесії в Node.js

Приклад використання Cookie в node.js:

```
const express = require('express'),
  app = express(),
  cookieParser = require('cookie-parser')

const host = '127.0.0.1'
const port = 7000

app.use(cookieParser('secret key'))

app.get('/get-cookie', (req, res) => {
  console.log('Cookie: ', req.cookies)
  res.send('Get Cookie')
})
```

```

app.get('/set-cookie', (req, res) => {
  res.cookie('token', '12345ABCDE')
  res.send('Set Cookie')
})

app.listen(port, host, () =>
  console.log(`Server listens http://${host}:${port}`)
)

```

Приклад використання сесій в node.js:

```

const express = require("express");
const app = express();
const session = require("express-session");
const cookieParser = require("cookie-parser");
const PORT = 4000;

app.use(cookieParser());

app.use(session({
  secret: process.env.SESSION_SECRET,
  saveUninitialized: true,
  resave: true
}));

app.get("/login", (req, res) => {
  req.session.user = { id: 1, name: 'Demo' };
  req.session.save();
  return res.send('Logged in');
});

app.listen(PORT, () => console.log(`Server at ${PORT}`));

```

8.7 npm та package.json

Менеджер пакетів npm (Node Package Manager) встановлюється разом з Node.js та є фактичним стандартом для роботи з JavaScript/TypeScript-залежностями. На 2026 рік реєстр npmjs.com містить понад 3 мільйони пакетів — це найбільший репозиторій програмних бібліотек у світі. Для серверних застосунків npm виконує ту саму роль, що й Composer для PHP: оголошення залежностей, їх встановлення та автозавантаження.

Основні команди npm

```

npm init           # створити новий проєкт (інтерактивно)
npm init -y        # створити проєкт зі значеннями за
замовчуванням
npm install <pkg>  # встановити пакет та додати до dependencies
npm install -D <pkg> # встановити як devDependency
npm run <script>    # виконати скрипт з package.json
npm <command>      # виконати CLI-утиліту без глобального
встановлення

```

```
npm update      # оновити встановлені пакети
npm audit       # перевірити вразливості безпеки
```

Структура package.json

Файл `package.json` — маніфест проєкту, що описує метадані, скрипти та залежності. Створюється командою `npm init` та автоматично оновлюється при встановленні пакетів. Приклад мінімального `package.json` для NestJS-проєкту:

```
{
  "name": "products-api",
  "version": "1.0.0",
  "description": "REST API для каталогу продукції",
  "scripts": {
    "start": "nest start",
    "start:dev": "nest start --watch",
    "build": "nest build",
    "test": "jest"
  },
  "dependencies": {
    "@nestjs/common": "^10.3.0",
    "@nestjs/core": "^10.3.0",
    "@prisma/client": "^5.10.0"
  },
  "devDependencies": {
    "@nestjs/cli": "^10.3.0",
    "prisma": "^5.10.0",
    "typescript": "^5.3.0"
  }
}
```

Ключові поля: `scripts` містить команди, які запускаються через `npm run <name>`; `dependencies` — залежності, необхідні в продакшні; `devDependencies` — інструменти для розробки (компілятори, тести, лінери), які не потрапляють у `production`-збірку.

Семантичне версіонування та lock-файл

Версії у форматі MAJOR.MINOR.PATCH (наприклад, 10.3.0). Обмеження `^10.3.0` дозволяє оновлення до 10.x, але не до 11.0; `~10.3.0` дозволяє лише патч-версії 10.3.x. Файл `package-lock.json` фіксує точні версії всіх залежностей (включно з транзитивними) і має бути закомічений у Git-репозиторій — це гарантує однакові залежності на всіх машинах команди та у CI/CD.

Питання та завдання для самоконтролю до підрозділу:

1. Чим відрізняються `dependencies` від `devDependencies`?
2. Що означає обмеження версії `^1.2.3`?
3. Навіщо потрібен файл `package-lock.json` і чи слід його комітити у Git?

8.8 TypeScript: основи для серверної розробки

TypeScript — мова програмування, що розширює JavaScript статичною типізацією. Розроблена компанією Microsoft, безкоштовна, з відкритим кодом (Apache 2.0). На 2026 рік TypeScript є стандартом серверної JavaScript-розробки в індустрії: Stack Overflow Survey 2025 показує, що 78% Node.js-проектів написані саме мовою TypeScript. Фреймворк NestJS, який ми вивчаємо у цьому розділі, повністю орієнтований на TypeScript — приклади у документації, генератор CLI і вся типізація API розраховані саме на TS.

Базовий синтаксис типів

```
// Примітивні типи
const name: string = "Іван";
const age: number = 25;
const isActive: boolean = true;
const tags: string[] = ["nestjs", "prisma"];

// Об'єднані типи (union types)
let id: number | string = 42;
id = "uuid-abc-123"; // також валідно

// Тип, що допускає null/undefined
let optional: string | null = null;

// Тип "any" — вимикає перевірку типів (не рекомендовано)
let unsafe: any = "будь-що";
```

Інтерфейси та типи

Інтерфейс описує форму об'єкта — набір полів та їх типи. Це базовий інструмент для створення DTO (Data Transfer Object) у NestJS:

```
interface Product {
  id: number;
  title: string;
  price: number;
  inStock: boolean;
  tags?: string[]; // ? — необов'язкове поле
}

function createProduct(data: Product): Product {
  // компілятор перевірить, що data має всі поля
  return {...data, id: Date.now() };
}
```

Класи з типізацією та декоратори

Класи у TypeScript мають типізовані властивості та методи. NestJS активно використовує декоратори (@Module, @Controller, @Injectable, @Get, @Post

тощо) — це функції, що додають метадані до класів та методів. Декоратори задаються префіксом `@` перед оголошенням і потребують увімкнення опції `experimentalDecorators` у `tsconfig.json`:

```
class ProductService {
  private products: Product[] = [];

  findAll(): Product[] {
    return this.products;
  }

  create(dto: Product): Product {
    this.products.push(dto);
    return dto;
  }
}
```

Компіляція та tsconfig.json

Файл `tsconfig.json` конфігурує компілятор TypeScript: цільову версію JavaScript (ES2022, ES2023), модульну систему (ESNext, CommonJS), суворість перевірок та шлях для скомпільованих файлів. NestJS-проект містить готовий `tsconfig.json` з оптимальними налаштуваннями для серверної розробки. Компіляція виконується автоматично командою `npm run build` або у режимі `watch` під час розробки (`npm run start:dev`).

Питання та завдання для самоконтролю до підрозділу:

1. Чим TypeScript відрізняється від JavaScript на рівні часу виконання та часу компіляції?
2. Що таке інтерфейс і навіщо його застосовувати?
3. Що таке декоратори та для чого вони використовуються у NestJS?

8.9 NestJS та Prisma ORM

Існуючий матеріал розділу (підрозділи 8.1–8.10) висвітлює базові поняття мови Node.js: подієвий цикл, асинхронність, модулі, Express. Для повноцінної розробки серверних застосунків цього недостатньо — реальні проекти використовують повностекові фреймворки, які забезпечують структуру, впровадження залежностей, валідацію та інтеграцію з базою даних.

У цій частині розглядається сучасний серверний фреймворк NestJS 10+ як основний інструмент для другого комп'ютерного практикуму, а також Prisma

ORM — type-safe засіб роботи з реляційними базами даних, що став індустрійним стандартом для Node.js-застосунків у 2025–2026 роках.

NestJS — сучасний серверний фреймворк для створення масштабованих, ефективних та надійних Node.js-застосунків. Розроблено Камілом Мисьлівцем (Kamil Myśliwiec), перший стабільний реліз у 2017 році. Фреймворк поширюється під ліцензією MIT, написаний мовою TypeScript і за замовчуванням орієнтований на TS-розробку.

Концептуально NestJS подібний до Angular на фронтенді та Spring Boot або Symfony на бекенді: модульна архітектура з впровадженням залежностей (Dependency Injection), декларативні маршрути через декоратори, чітке розділення відповідальності між модулями, контролерами та сервісами. На 2026 рік фреймворк використовується у Adidas, Roche, Trilon, Decathlon — це робить його практично затребуваним для випускників.

Чому саме NestJS, а не Express

Express — мінімалістичний фреймворк, що надає лише низькорівневі примітиви (router, middleware). Архітектурні рішення (структура файлів, DI, валідація, тести) розробник приймає сам. Для невеликих скриптів це зручно, але для серверних API з кількома десятками маршрутів швидко стає проблемою: код дублюється, тести складно писати, рефакторинг болючий.

NestJS вирішує ці проблеми «з коробки», накладаючи структуру: кожна частина застосунку оформлюється як модуль (@Module), HTTP-маршрути живуть у контролерах (@Controller), бізнес-логіка — у сервісах (@Injectable). Під капотом NestJS використовує Express або Fastify — програміст отримує і високорівневу архітектуру, і повну сумісність з екосистемою Express middleware.

Архітектура NestJS-застосунку

Базовою одиницею NestJS-застосунку є модуль. Модуль — це клас, прикрашений декоратором @Module, який оголошує які контролери, сервіси та інші модулі належать цій логічній області. Кореневий модуль AppModule імпортує функціональні модулі (наприклад, ProductsModule, UsersModule, AuthModule), кожен з яких самодостатній і може бути перенесений у інший проєкт.

Усередині модуля живуть три типи елементів: контролери (приймають HTTP-запити та повертають відповіді), сервіси (бізнес-логіка, робота з БД, інтеграції) та провайдери (загальна категорія для всього, що може бути впроваджено через DI). Кожен елемент описується класом з відповідним декоратором.

Таблиця 8.2 – Порівняння з Symfony і NestJS

Концепція	Symfony	NestJS
HTTP-роутинг	Атрибут #[Route('/path')]	Декоратор @Controller('path') + @Get('/')
Бізнес-логіка	Service (POPO-клас)	Service (клас з @Injectable())
Впровадження залежностей	Service Container (autowiring)	DI Container (autowiring)
ORM (типово)	Doctrine ORM	Prisma або TypeORM
Валідація	Symfony Validator + #[Assert\NotBlank]	class-validator + @IsString()
Стандарт відповіді	JsonResponse	Авто-серіалізація об'єкта в JSON

Питання та завдання для самоконтролю до підрозділу:

1. Чим NestJS концептуально відрізняється від Express?
2. Що таке модуль у NestJS?
3. Які елементи може містити модуль?

8.10 Створення проєкту та структура NestJS

NestJS постачається з власною CLI-утилітою, яка автоматизує створення проєкту, генерацію контролерів, сервісів та модулів. CLI знає про конвенції фреймворку та одразу формує правильну структуру файлів.

Встановлення CLI та створення проєкту

```
# Глобальне встановлення CLI (виконується одного разу)
npm i -g @nestjs/cli

# Створення нового проєкту
nest new my-project

# CLI запитатиме менеджер пакетів (npm/yarn/pnpm)
# та згенерує проєкт з усіма залежностями

# Запуск проєкту в режимі розробки (з автоперезапуском)
cd my-project
npm run start:dev
```

Після виконання `nest new my-project` буде створено директорію з готовою структурою файлів, налаштованим TypeScript, Jest для тестів та ESLint для літінгу. Сервер запускається на `http://localhost:3000` та одразу відповідає GET-запит на корінь.

Структура каталогу src/

```
my-project/
├── src/
│   ├── main.ts                # точка входу — створення Nest-
застосунку
│   ├── app.module.ts          # кореневий модуль
│   ├── app.controller.ts      # приклад контролера
│   ├── app.service.ts         # приклад сервісу
│   └── app.controller.spec.ts # юніт-тест контролера
├── test/                      # e2e-тести
├── package.json
├── tsconfig.json
└── nest-cli.json              # конфігурація CLI
```

Точка входу — main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors(); // дозвіл CORS для SPA-клієнтів
  app.setGlobalPrefix('api'); // префікс /api для всіх маршрутів
  await app.listen(3000);
}
bootstrap();
```

Кореневий модуль app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [], // інші модулі цього застосунку
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Декоратор `@Module` реєструє метадані: які модулі імпортуються (їх провайдери стають доступними), які контролери оброблюють HTTP-запити та які провайдери (сервіси, репозиторії) надає цей модуль для DI. Виконуючи `nest`

g module products, CLI створить файл products.module.ts з аналогічною структурою — так нарощується нова функціональність.

Питання та завдання для самоконтролю до підрозділу:

1. Як створити новий NestJS-проект?
2. Яку роль виконує main.ts?
3. Що означають поля imports/controllers/providers у декораторі @Module?

8.11 REST API у NestJS: контролери та сервіси

Контролер у NestJS відповідає за прийом HTTP-запитів та повернення відповідей. Кожен контролер реєструється у модулі та обробляє запити за певним базовим шляхом. Сервіс містить бізнес-логіку, ізольовану від HTTP-шару — це робить код легко тестованим та переносимим.

Генерація ресурсу одною командою

```
nest generate resource products
# або скорочено:
nest g resource products

# CLI запропонує тип REST API та чи генерувати CRUD entry points
# і створить products.module.ts, products.controller.ts,
# products.service.ts, products.controller.spec.ts,
# products.service.spec.ts, dto/create-product.dto.ts
# та entities/product.entity.ts
```

Контролер з повним CRUD

```
import { Controller, Get, Post, Put, Delete, Body, Param } from
 '@nestjs/common';
import { ProductsService } from '../products.service';
import { CreateProductDto } from '../dto/create-product.dto';

@Controller('products')
export class ProductsController {
  constructor(private readonly productsService: ProductsService) {}

  @Get()
  findAll() {
    return this.productsService.findAll();
  }

  @Get('/:id')
  findOne(@Param('id') id: string) {
    return this.productsService.findOne(+id);
  }
}
```

```

@Post()
create(@Body() dto: CreateProductDto) {
  return this.productsService.create(dto);
}

@Put('/:id')
update(@Param('id') id: string, @Body() dto: CreateProductDto) {
  return this.productsService.update(+id, dto);
}

@Delete('/:id')
remove(@Param('id') id: string) {
  return this.productsService.remove(+id);
}
}

```

Декоратори `@Get`, `@Post`, `@Put`, `@Delete` мають метод на HTTP-дієслово; `@Param('id')` витягує параметр зі шляху URL; `@Body()` дезеріалізує тіло запиту у DTO-об'єкт. Конструктор класу автоматично отримує сервіс через DI — Nest самостійно створить його екземпляр та передасть.

Відповідний сервіс

```

import { Injectable, NotFoundException } from '@nestjs/common';

@Injectable()
export class ProductsService {
  private products = [];
  private nextId = 1;

  findAll() {
    return this.products;
  }

  findOne(id: number) {
    const product = this.products.find(p => p.id === id);
    if (!product) {
      throw new NotFoundException(`Продукт ${id} не знайдено`);
    }
    return product;
  }

  create(dto) {
    const product = { id: this.nextId++, ...dto };
    this.products.push(product);
    return product;
  }
}

```

Приклади запитів та відповідей

```

# Запит:
POST /api/products HTTP/1.1

```

```

Content-Type: application/json
{ "title": "Ноутбук", "price": 24990.50 }

# Відповідь:
201 Created
Content-Type: application/json
{ "id": 1, "title": "Ноутбук", "price": 24990.50 }

# Запит:
GET /api/products/1 HTTP/1.1

# Відповідь:
200 OK
{ "id": 1, "title": "Ноутбук", "price": 24990.50 }

```

Якщо ресурс не знайдено, NestJS автоматично формує JSON-відповідь з кодом 404 на основі викинутого `NotFoundException` — це стандартна поведінка вбудованого фільтра винятків.

Питання та завдання для самоконтролю до підрозділу:

1. Які декоратори використовуються для маршрутизації у NestJS?
2. Як отримати параметр з URL та тіло запиту?
3. Як NestJS трансформує винятки у HTTP-відповіді?

8.12 Робота з базою даних: Prisma ORM

Prisma — сучасний type-safe ORM для Node.js та TypeScript. Розроблено компанією Prisma Data Inc., перший стабільний реліз у 2021 році, з 2023 року використовується у виробничих проєктах Vercel, Netflix, Adobe. На відміну від класичних ORM (Doctrine у Symfony, TypeORM у Node.js), Prisma використовує декларативну схему та автогенерований клієнт — це дає програмістові 100% type-safety, а адміністраторові БД — автоматичні SQL-міграції з diff схеми.

У цьому посібнику ми зупиняємось саме на Prisma (а не на TypeORM, який також інтегрується з NestJS) тому що декларативна модель `schema.prisma` набагато наочніша для студентів, генерований клієнт виключає цілий клас помилок типізації, а команди `prisma migrate` забезпечують безпечне еволюціонування схеми БД.

Встановлення та ініціалізація

```

# Встановлення CLI та клієнта
npm install prisma --save-dev

```

```
npm install @prisma/client
```

```
# Ініціалізація: створює директорію prisma/ зі schema.prisma
та.env
npx prisma init --datasource-provider mysql
```

Файл `schema.prisma` — єдине джерело правди

`schema.prisma` описує підключення до БД, генератор клієнта та моделі даних. Це декларативна мова, специфічна для Prisma, дуже коротка і виразна:

```
// prisma/schema.prisma

datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model Product {
  id          Int          @id @default(autoincrement())
  title       String       @db.VarChar(255)
  price       Decimal      @db.Decimal(10, 2)
  inStock     Boolean      @default(true)
  tags        Json?
  createdAt   DateTime     @default(now())
  updatedAt   DateTime     @updatedAt
}
```

Атрибути `@id`, `@default`, `@db.VarChar(255)` визначають властивості колонок; рядок `env("DATABASE_URL")` посиляється на змінну середовища у файлі `.env`, де зберігається DSN для підключення (наприклад, `mysql://root:secret@localhost:3306/products_db`). Зберігання DSN у `.env`, а не в коді, відповідає принципу 12-Factor App.

Міграції — автоматично з `diff` схеми

```
# Створити нову міграцію та одразу застосувати її до БД
npx prisma migrate dev --name init

# Згенерувати клієнт (зазвичай виконується автоматично після
migrate)
npx prisma generate

# Застосувати міграції у production-середовищі
npx prisma migrate deploy

# Відкрити Prisma Studio — графічний інтерфейс для перегляду БД
npx prisma studio
```

При першому запуску `prisma migrate dev` створить директорію `prisma/migrations/` з SQL-файлом для початкової схеми. Подальші зміни моделей (додавання полів, нових моделей, зміна типів) автоматично призводять до генерації нових міграцій з відповідним SQL ALTER/CREATE/DROP. Програміст не пише SQL вручну, але може переглянути його та відредагувати перед застосуванням.

PrismaService — інтеграція з NestJS

Для зручного використання Prisma у NestJS створюється сервіс-обгортка, яка успадковує `PrismaClient` та керує життєвим циклом підключення:

```
// src/prisma/prisma.service.ts
import { Injectable, OnModuleInit, OnModuleDestroy } from
 '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient
  implements OnModuleInit, OnModuleDestroy {

  async onModuleInit() {
    await this.$connect();
  }

  async onModuleDestroy() {
    await this.$disconnect();
  }
}
```

CRUD-операції через Prisma

```
// src/products/products.service.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';

@Injectable()
export class ProductsService {
  constructor(private readonly prisma: PrismaService) {}

  findAll() {
    return this.prisma.product.findMany();
  }

  findOne(id: number) {
    return this.prisma.product.findUnique({ where: { id } });
  }

  create(data) {
    return this.prisma.product.create({ data });
  }

  update(id: number, data) {
    return this.prisma.product.update({ where: { id }, data });
  }
}
```

```

    }

    remove(id: number) {
      return this.prisma.product.delete({ where: { id } });
    }
  }
}

```

Усі методи Prisma-клієнта повертають Promise з повністю типізованою відповіддю: IDE підказує доступні поля, помилка типу буде помічена під час компіляції. Опція `include` дозволяє завантажити пов'язані сутності одним запитом (Prisma автоматично оптимізує SQL з JOIN), а `select` — обмежити список полів у відповіді.

Таблиця 8.3 – Порівняння з Doctrine ORM (Symfony)

Концепція	Doctrine ORM	Prisma
Опис моделі	Атрибути PHP у класі сутності	Окремий файл <code>schema.prisma</code>
Типобезпека	Сильна (PHP 8.x types)	Повна (згенерований TS-клієнт)
Міграції	<code>doctrine:migrations:diff</code>	<code>prisma migrate dev</code>
Запит за ID	<code>\$repo->find(\$id)</code>	<code>prisma.product.findUnique({ where: { id } })</code>
Збереження	<code>persist + flush</code>	<code>prisma.product.create({ data })</code>
Завантаження зв'язків	EAGER/LAZY у конфігурації	<code>include / select</code> у запиті

Питання та завдання для самоконтролю до підрозділу:

1. Що таке `schema.prisma` і яку роль вона виконує?
2. Як створити та застосувати міграцію з Prisma?
3. Чим відрізняються методи `findUnique` та `findFirst`?
4. Як отримати продукт разом з пов'язаними тегами одним запитом?

8.13 Валідація та DTO у NestJS

DTO (Data Transfer Object) — клас, що описує структуру вхідних даних запиту. У NestJS DTO використовуються для типізації параметра `@Body()` та автоматичної валідації. Замість того, щоб писати `if`-перевірки у контролері, ми описуємо правила декораторами на властивостях DTO — фреймворк сам перевірить вхідні дані до того, як виклик дійде до контролера.

Встановлення та глобальне підключення

```

npm install class-validator class-transformer
// src/main.ts
import { NestFactory } from '@nestjs/core';
import { ValidationPipe } from '@nestjs/common';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true,           // прибрати поля, що не оголошені
    forbidNonWhitelisted: true, // повернути 400, якщо є зайві
    transform: true,          // автоматично перетворювати типи
  }));
  await app.listen(3000);
}
bootstrap();

```

Створення DTO з декораторами

```

// src/products/dto/create-product.dto.ts
import { IsString, IsNumber, IsBoolean, IsOptional,
  IsArray, MinLength, IsPositive } from 'class-validator';

export class CreateProductDto {
  @IsString()
  @MinLength(2, { message: 'Назва має містити мінімум 2 символи' })
  title: string;

  @IsNumber()
  @IsPositive()
  price: number;

  @IsBoolean()
  @IsOptional()
  inStock?: boolean;

  @IsArray()
  @IsString({ each: true })
  @IsOptional()
  tags?: string[];
}

```

При спробі надіслати POST-запит з невалідними даними (наприклад, ціна = -5 або відсутня назва) NestJS автоматично поверне відповідь 400 Bad Request з

масивом помилок:

```

# Запит з невалідними даними:
POST /api/products
{ "title": "X", "price": -5 }

# Відповідь:
400 Bad Request
{
  "statusCode": 400,

```

```

"message": [
  "Назва має містити мінімум 2 символи",
  "price must be a positive number"
],
"error": "Bad Request"
}

```

Таблиця 8.4 – Порівняння з Symfony Validator

Правило	Symfony Validator	class-validator
Не порожнє	<code>#[Assert\NotBlank]</code>	<code>@IsNotEmpty()</code>
Рядок мін. 2 символи	<code>#[Assert\Length(min: 2)]</code>	<code>@MinLength(2)</code>
Додатне число	<code>#[Assert\Positive]</code>	<code>@IsPositive()</code>
Email-адреса	<code>#[Assert\Email]</code>	<code>@IsEmail()</code>
Опціональне поле	<code>#[Assert\Optional]</code>	<code>@IsOptional()</code>

Питання та завдання для самоконтролю до підрозділу:

1. Що таке DTO у NestJS?
2. Як підключити валідацію глобально?
3. Які декоратори використовуються для перевірки рядків та чисел?
4. Як обмежити поля запиту лише оголошеними у DTO?

8.14 Гварди, перехоплювачі та JWT-аутентифікація

Окрім контролерів, сервісів і DTO, NestJS пропонує два спеціалізовані компоненти для перехресних задач (cross-cutting concerns) — гварди (guards) та перехоплювачі (interceptors). Гвард ухвалює рішення «пускати запит до контролера чи ні» (типово — аутентифікація та авторизація). Перехоплювач огортає виклик контролера й може змінювати запит до виклику, відповідь — після виклику, фіксувати журнал, кешувати результат. Обидва — це класи з декоратором `@Injectable()`, які реалізують відповідний інтерфейс з одним обов'язковим методом.

Гварди (Guards)

Гвард — клас з декоратором `@Injectable()`, який реалізує інтерфейс `CanActivate` з єдиним методом `canActivate(context)`. Метод повертає `true` (пропустити запит далі) або `false` / викидає виняток (заблокувати запит).

Контекст `ExecutionContext` дає доступ до базового HTTP-запиту, метаданих маршруту та аргументів обробника.

```
// src/auth/auth.guard.ts
import { CanActivate, ExecutionContext, Injectable,
        UnauthorizedException } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private readonly jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = request.headers['authorization']?.split(' ')[1];
    if (!token) throw new UnauthorizedException('Токен відсутній');
    try {
      request.user = await this.jwtService.verifyAsync(token);
      return true;
    } catch {
      throw new UnauthorizedException('Невалідний токен');
    }
  }
}
```

Гвард застосовується декоратором `@UseGuards()` — на рівні методу, контролера або глобально. У прикладі нижче GET-список доступний усім, а POST доступний лише авторизованим користувачам:

```
@Controller('products')
export class ProductsController {
  @Get() findAll() { /* публічний доступ */ }

  @UseGuards(AuthGuard)
  @Post() create(@Body() dto: CreateProductDto) { /* лише авторизовані */ }
}
```

Перехоплювачі (Interceptors)

Перехоплювач реалізує інтерфейс `NestInterceptor` з методом `intercept(context, next)`. Перед `next.handle()` виконується код «до контролера»; після — через RxJS-оператори (`tap`, `map`) — код «після відповіді». Типові сценарії: журналювання часу обробки, трансформація відповіді, кешування, серіалізація.

```
// src/common/logging.interceptor.ts
import { CallHandler, ExecutionContext, Injectable,
        NestInterceptor } from '@nestjs/common';
import { Observable, tap } from 'rxjs';

@Injectable()
```

```
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler):
  Observable<any> {
    const start = Date.now();
    const { method, url } = context.switchToHttp().getRequest();
    return next.handle().pipe(
      tap(() => console.log(`${method} ${url} - ${Date.now() -
start}ms`)),
    );
  }
}
```

Підключення глобально у main.ts: `app.useGlobalInterceptors(new LoggingInterceptor())`. Як і гварди, перехоплювачі можуть застосовуватися декоратором `@UseInterceptors()` на рівні окремого методу або контролера.

JWT-ауθενфікація з @nestjs/jwt

JSON Web Token (JWT) — компактний токен з трьох частин (header.payload.signature), підписаний секретним ключем. Сервер видає JWT при успішному вході; клієнт надсилає його у заголовок `Authorization: Bearer <token>` з кожним наступним запитом. Сервер верифікує підпис, читає payload (наприклад, `userId`, `role`) і допускає або відхиляє запит. Перевага перед серверними сесіями — відсутність серверного стану, що робить JWT природним для горизонтально масштабованих сервісів.

Встановлення

```
npm install @nestjs/jwt
```

Реєстрація модуля з секретом і часом життя токена:

```
// src/auth/auth.module.ts
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { AuthService } from './auth.service';
import { AuthGuard } from './auth.guard';

@Module({
  imports: [
    JwtModule.register({
      global: true,
      secret: process.env.JWT_SECRET,
      signOptions: { expiresIn: '1h' },
    }),
  ],
  providers: [AuthService, AuthGuard],
  exports: [AuthService, AuthGuard],
})
export class AuthModule {}
```

Сервіс, який видає токен після перевірки облікових даних:

```
// src/auth/auth.service.ts
import { Injectable, UnauthorizedException } from
 '@nestjs/common';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {
  constructor(private readonly jwt: JwtService) {}

  async login(email: string, password: string) {
    // Спрощено: реальна перевірка пароля — через bcrypt і
    UsersService.
    const payload = { sub: 1, email, role: 'admin' };
    return { access_token: await this.jwt.signAsync(payload) };
  }
}
```

Секрет `JWT_SECRET` зберігається у файлі `.env` та ніколи не комітється в Git — це базова вимога безпеки (OWASP API Security 2023, A02:2023 Broken Authentication). У продакшні замість симетричного HS256 рекомендується використовувати асиметричний RS256 з ротацією ключів.

Питання для самоконтролю до підрозділу: Чим гвард відрізняється від перехоплювача? Який метод реалізує клас гварда та що повертає? Як декоратор `@UseGuards()` застосовується до окремого методу контролера? Чим JWT відрізняється від серверних сесій з точки зору масштабування? Чому секрет `JWT_SECRET` зберігається у `.env`, а не в коді?

8.15 Питання та завдання для самоконтролю до Розділу 8

1. Чим відрізняється модель виконання Node.js від традиційного багатопоточного PHP-сервера?
2. Які типи задач погано підходять для одного потоку Node.js і як їх обробляти (`worker_threads`)?
3. Що таке `npm` та для чого використовується файл `package.json`?
4. Чим відрізняються поля `dependencies` та `devDependencies` у `package.json`?
5. Що таке TypeScript та які переваги типобезпеки на бекенді?
6. Що таке інтерфейс у TypeScript і як його застосовувати для DTO?
7. Які декоратори використовуються для маршрутизації у NestJS?
8. Що таке модуль у NestJS і які елементи він може містити?
9. Як NestJS реалізує впровадження залежностей?

10. Як створити новий ресурс (контролер + сервіс + DTO) однією командою?
11. Що таке `schema.prisma` і чому Prisma називає її «єдиним джерелом правди»?
12. Як виконати міграцію у Prisma та що таке `prisma generate`?
13. Чим Prisma переважно обирається замість TypeORM (декларативна схема, `type-safety`, автоматичні міграції)?
14. Що таке DTO і навіщо його застосовувати у контролерах?
15. Як підключити глобальну валідацію у NestJS та які опції `ValidationPipe` найважливіші?
16. Як NestJS повертає помилки валідації клієнту та який HTTP-код використовується?
17. Чим декоратори `class-validator` (наприклад, `@IsString`, `@MinLength`) відрізняються від атрибутів `Symfony Validator`?
18. Чим відрізняються підходи до валідації в `Symfony` та `NestJS` у разі невалідних даних запиту?
19. Що таке цикл подій (`event loop`) і чому він є серцем неблокуючої моделі `Node.js`?
20. Чому `Node.js` погано підходить для CPU-ємних завдань і як цю проблему обходять через `Worker Threads`?
21. Яка різниця між полями `dependencies` та `devDependencies` у файлі `package.json`?
22. Що таке декоратори у NestJS (`@Controller`, `@Injectable`, `@Get`) і яку роль вони виконують?
23. Чим Prisma ORM відрізняється від `Doctrine ORM` за способом визначення схеми бази даних?
24. Що таке `Guard` у NestJS і як він використовується для реалізації JWT-аутентифікації?

Лекція 9. РОЗГОРТАННЯ ВЕБ-ДОДАТКІВ У МЕРЕЖІ

План лекції:

1. Вступ
2. Розгортання простих PHP-сайтів через FTP
3. Контейнеризація з Docker
4. GitLab CI
5. GitHub Actions

Ключові слова: FTP, Docker, контейнеризація, контроль версій.

9.1 Вступ

Сучасне розгортання серверних веб-додатків відбувається не через копіювання файлів на хостинг, а через контейнеризацію — пакування додатка з усіма його залежностями (інтерпретатором мови, бібліотеками, конфігурацією) у самодостатню одиницю — контейнер. Контейнер запускається однаково на ноутбучі розробника, у пайплайні CI, у production-кластері — це дозволяє забезпечити правило «build once, run anywhere» і повністю виключити ситуацію «у мене все працює».

Цей розділ описує промисловий стек розгортання 2026 року: Docker як платформа контейнеризації, Docker Compose як інструмент оркестрації багатоконтейнерного застосунку та системи неперервної інтеграції (GitLab CI, GitHub Actions), які автоматизують збірку, тестування та публікацію образів. Третій комп'ютерний практикум курсу безпосередньо ґрунтується на цьому матеріалі: ви контейнеризуєте обидва сервіси (Symfony з комп'ютерного практикуму № 1 і NestJS+Prisma з комп'ютерного практикуму № 2) та налаштуєте CI-конвеєр у GitHub Actions.

9.2 Розгортання простих PHP-сайтів через FTP

Найпростіший сценарій розгортання — статичний PHP-сайт на shared-хостингу — досі іноді виконується через FTP-клієнт (FileZilla, WinSCP, Cyberduck): ви вручну копіюєте файли проєкту на сервер хостинг-провайдера.

Це підхід 2000-х років: неавтоматизовано, без версіонування, без можливості швидко відкотитися. У сучасних професійних проєктах FTP практично не використовується — його повністю витіснили Git-based deploy, контейнеризація та CI/CD-конвеєри. У цьому посібнику FTP розглядається лише для повноти картини: ваш бізнес-стек (комп'ютерні практикуми № 1–3) ґрунтується на Docker, описаному далі.

9.3 Контейнеризація з Docker

Docker — найпоширеніша платформа контейнеризації 2026 року, де-факто стандарт для пакування серверних застосунків. Розроблена компанією Docker Inc., перший стабільний реліз у 2013 році. Docker дозволяє: 1) ізолювати додаток від хост-ОС, 2) гарантувати ідентичність середовища у розробці й продакшні, 3) розгортати додаток однією командою, 4) масштабувати горизонтально (запустити 10 однакових контейнерів) без зайвих зусиль.

Основні поняття Docker

Образ (image) — read-only шаблон, з якого створюються контейнери. Образ складається з шарів (layers): кожна інструкція у Dockerfile додає новий шар. Шари кешуються — повторна збірка образу при незмінному коді відбувається миттєво. Образи зберігаються у реєстрі (registry): публічний Docker Hub (hub.docker.com), приватні GitHub Container Registry, GitLab Container Registry, AWS ECR тощо.

Контейнер (container) — запущений екземпляр образу. Контейнерів з одного образу можна запустити скільки завгодно; вони ізольовані між собою та від хост-ОС. Контейнер по суті — це процес із власним простором імен (namespaces) та обмеженнями ресурсів (cgroups), що накладається ядром Linux.

Том (volume) — спосіб зберегти дані поза контейнером. Контейнер ефемерний — при перезапуску всі зміни всередині нього зникають. Дані БД, завантажені користувачами файли тощо мають жити у томах, які монтуються у файлову систему контейнера.

Мережа (network) — Docker створює віртуальні мережі для контейнерів, що дозволяє їм спілкуватися за іменем сервісу (наприклад, додаток підключається до БД через `host=mysql`, `port=3306`, а не через IP). Це базова функціональність Docker Compose.

Dockerfile — текстовий файл з інструкціями, з якого Docker будує образ.

Найпростіший Dockerfile для PHP-додатку 2026 року:

```
FROM php:8.5-fpm-alpine
WORKDIR /var/www/html
COPY composer.json composer.lock ./
RUN composer install --no-dev --optimize-autoloader
COPY . .
EXPOSE 9000
CMD ["php-fpm"]
```

Ключова відмінність від v7-епохи (FROM `php:7.4-apache`): ми використовуємо PHP 8.5 (актуальна гілка на момент 2026 року, реліз — листопад 2025), `php-fpm` (FastCGI Process Manager — стандарт для інтеграції з Nginx у production), варіант `-alpine` (мінімальний образ ~30 МБ проти ~400 МБ у full-варіанті).

Dockerfile та multi-stage build

Multi-stage build — техніка, що дозволяє використати кілька проміжних образів для збірки, але у фінальному образі залишити лише виконувані файли без зайвих інструментів. Це різко зменшує розмір production-образу та поверхню атаки.

Приклад Dockerfile для Symfony-проєкту з multi-stage build:

```
# === Stage 1: builder ===
FROM php:8.5-cli-alpine AS builder
WORKDIR /app

# Системні залежності для composer та розширень PHP
RUN apk add --no-cache git unzip libzip-dev \
    && docker-php-ext-install zip pdo pdo_mysql

COPY --from=composer:2 /usr/bin/composer /usr/bin/composer
COPY composer.json composer.lock ./
RUN composer install --no-dev --no-scripts --optimize-autoloader

COPY . .
RUN composer dump-autoload --optimize

# === Stage 2: production ===
FROM php:8.5-fpm-alpine
WORKDIR /var/www/html
```

```

RUN docker-php-ext-install pdo pdo_mysql opcache
COPY --from=builder /app /var/www/html

EXPOSE 9000
CMD ["php-fpm"]

```

Етап `builder` містить інструменти збирання (`composer`, `git`, `dev-розширення`); етап `production` — лише виконуваний PHP-код проєкту. Розмір `production`-образу виходить ~50 МБ замість ~400 МБ у однофазному варіанті.

Приклад `Dockerfile` для NestJS-проєкту:

```

# === Stage 1: builder ===
FROM node:24-alpine AS builder
WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build          # компілює TS у dist/
RUN npm prune --omit=dev   # видаляє devDependencies

# === Stage 2: production ===
FROM node:24-alpine
WORKDIR /app
ENV NODE_ENV=production

COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package.json ./package.json

EXPOSE 3000
CMD ["node", "dist/main.js"]

```

Ключові директиви `Dockerfile`: `FROM` (базовий образ), `WORKDIR` (робочий каталог), `COPY` (копіювання файлів з хоста у контейнер), `RUN` (виконати команду під час збірки — кешується), `CMD` (команда, яка виконується при запуску контейнера — НЕ кешується), `EXPOSE` (документує, який порт слухає додаток), `ENV` (змінна середовища).

Файл `.dockerignore` (поряд з `Dockerfile`) виключає файли з контексту збірки — `node_modules`, `.git`, `env*`, `vendor/` тощо. Без нього збірка завантажує мегабайти непотрібного, шари кешуються погано і фінальний образ роздувається.

Особливо критично виключити саме `vendor/` (для PHP) та `node_modules/` (для Node.js). Логіка обох наших `Dockerfile` така: спочатку `COPY`

composer.json/package*.json + RUN install — це створює чисті залежності всередині Linux-контейнера; потім COPY . . — копіює решту коду. Якщо у локальному vendor/ або node_modules/ на хост-машині лежать БІНАРНІ розширення, скомпільовані під вашу ОС (Windows / macOS), команда COPY . . перетре щойно зібрані Linux-бінарники цими несумісними файлами — і контейнер впаде при першому ж require. Файл.dockerignore унеможливорює цю помилку.

Docker Compose

Реальний застосунок складається з кількох контейнерів: бекенд + БД + кеш + веб-сервер тощо. Запускати їх вручну через docker run неможливо — потрібно пам'ятати порти, тома, залежності, мережу. Docker Compose описує всю систему у декларативному файлі docker-compose.yml; одна команда docker compose up запускає весь стек.

Приклад docker-compose.yml для Symfony + MySQL + Nginx:

```
services:
  app:
    build:..
    image: products-symfony
    environment:
      DATABASE_URL: mysql://root:secret@db:3306/products
    depends_on:
      db:
        condition: service_healthy

  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: products
    volumes:
      - db_data:/var/lib/mysql
    ports:
      - "3306:3306"
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 5s
      retries: 10

  nginx:
    image: nginx:1.27-alpine
    ports:
      - "8080:80"
    volumes:
      - ./docker/nginx.conf:/etc/nginx/conf.d/default.conf:ro
    depends_on:
```

```

- app

volumes:
  db_data:

```

Розшифровка ключів: `services` описує перелік контейнерів (тут `app`, `db`, `nginx`); `build` вказує шлях до `Dockerfile` для збірки; `image` — готовий образ з реєстру; `environment` задає змінні середовища всередині контейнера; `depends_on` забезпечує порядок запуску; `volumes` — томи для збереження даних; `ports` — пробрасує порт хоста у контейнер (формат `host:container`).

Приклад `docker-compose.yml` для NestJS + Prisma + MySQL (комп'ютерний практикум № 2):

```

services:
  api:
    build: .
    image: products-nestjs
    ports:
      - "3000:3000"
    environment:
      DATABASE_URL: mysql://root:secret@db:3306/products
      JWT_SECRET: ${JWT_SECRET}
    depends_on:
      db:
        condition: service_healthy

  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: products
    volumes:
      - db_data:/var/lib/mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 5s
      retries: 10

volumes:
  db_data:

```

У цьому прикладі ми додали `healthcheck` — Docker Compose чекатиме, поки MySQL не пройде `ping`, і лише потім стартуватиме API. Без `healthcheck` `depends_on` лише гарантує порядок СТАРТУ контейнерів, а не ГОТОВНІСТЬ сервісу.

Основні команди Docker

Нижче — команди, які потрібно знати на пам'ять. Більшість виконуються у корені проєкту, де лежить `Dockerfile` або `docker-compose.yml`.

Таблиця 9.1 – Основні команди Docker

Команда	Що робить
<code>docker build -t my-image.</code>	Побудувати образ з поточного Dockerfile та позначити тегом

продовження таблиці 9.1

<code>docker run -p 8080:80 my-image</code>	Запустити контейнер, прокинути порт 8080→80
<code>docker ps</code>	Список запущених контейнерів
<code>docker ps -a</code>	Усі контейнери, включаючи зупинені
<code>docker logs -f <container></code>	Слідкувати за журналом контейнера у реальному часі
<code>docker exec -it <container> sh</code>	Зайти у запущений контейнер інтерактивно
<code>docker stop <container></code>	М'яко зупинити контейнер
<code>docker rm <container></code>	Видалити зупинений контейнер
<code>docker images</code>	Список локальних образів
<code>docker rmi <image></code>	Видалити образ
<code>docker compose up -d</code>	Запустити весь стек у фоні (detached)
<code>docker compose down</code>	Зупинити та видалити всі контейнери стека
<code>docker compose down -v</code>	Те саме + видалити томи (УВАГА: втрата даних)
<code>docker compose build</code>	Перебудувати образи стека
<code>docker compose logs -f api</code>	Дивитися логи конкретного сервісу
<code>docker compose exec api sh</code>	Зайти у контейнер сервісу api

Зверніть увагу: `docker compose` (через пробіл) — нова форма команди як плагіну Docker CLI; стара форма `docker-compose` (через дефіс) — окремий бінарник, який поступово виходить з ужитку. У всіх нових документаціях та СІ-конфігураціях використовується нова форма.

Змінні середовища та .env-файли

У сучасних застосунках ВСЯ конфігурація, що залежить від середовища (DB-DSN, API-ключі, JWT-секрет, режим `debug` тощо) виноситься у змінні середовища. Це ключовий принцип методології 12-Factor App (<https://12factor.net>) — стандарту для cloud-native додатків.

Основна ідея: один і той самий образ запускається у `dev`, `staging`, `production` БЕЗ ПЕРЕЗБОРКИ. Різні середовища отримують різні значення змінних

середовища. Жодних конфігураційних файлів типу `config.dev.php` / `config.prod.php` — лише один шаблон, що читає `env`.

Структура `env`-файлів за конвенцією `Symfony` та `NestJS`:

```
#.env — значення за замовчуванням, КОМИТИТЬСЯ у Git
APP_ENV=dev
DATABASE_URL=mysql://root:secret@127.0.0.1:3306/products

#.env.local — локальні overrides, НЕ КОМИТИТЬСЯ
DATABASE_URL=mysql://my-user:my-pass@127.0.0.1:3306/my-db

#.env.production — production-значення (часто з Vault/SSM)
APP_ENV=prod
DATABASE_URL=mysql://prod-
user:${DB_PASSWORD}@db.example.com:3306/products
```

Файл `env.local` та подібні з секретами **ОБОВ'ЯЗКОВО** додають у `.gitignore`.

Закомічені секрети — типова причина витоків і безпеки-інцидентів. У промислових середовищах секрети зберігаються у системах типу `HashiCorp Vault`, `AWS Secrets Manager`, `Doppler`, `GitHub Secrets` — і інжектуються в контейнер під час запуску, а не читаються з файлів.

`Docker Compose` автоматично читає файл `env` у поточному каталозі та підставляє `${VAR}`-патерни у `docker-compose.yml`. Приклад використання: `${JWT_SECRET}` у розділі `environment` — `Docker Compose` замінить це на значення з `env` або відмовить у запуску, якщо змінна не визначена (з прапором `-env-file`).

9.4 GitLab CI

`GitLab CI` — система неперервної інтеграції, інтегрована в платформу `GitLab`. Конфігурація живе у файлі `.gitlab-ci.yml` у корені репозиторію — після кожного `push` `GitLab` автоматично запускає пайплайн (послідовність задач): літери, тести, збірка `Docker`-образу, публікація у реєстр, розгортання.

Приклад `.gitlab-ci.yml` для `NestJS`-проєкту з `Docker`-збіркою:

```
stages:
  - test
  - build

test:
  stage: test
  image: node:24-alpine
  script:
```

```

- npm ci
- npm run lint
- npm test

build-image:
  stage: build
  image: docker:24
  services:
    - docker:24-dind
  variables:
    IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
      $CI_REGISTRY
    - docker build -t $IMAGE .
    - docker push $IMAGE
  only:
    - main

```

Етапи (stages) виконуються послідовно; задачі всередині одного етапу — паралельно. Образ `docker:24-dind` (Docker-in-Docker) дозволяє виконувати `docker build` усередині `job`. Змінні з префіксом `CI_` (`CI_COMMIT_SHA`, `CI_REGISTRY_*`) надає GitLab автоматично; секрети додаються через Settings → CI/CD → Variables (НЕ зберігайте їх у `.gitlab-ci.yml`).

9.5 GitHub Actions

GitHub Actions — аналог GitLab CI для GitHub. У 2026 році це найпоширеніша платформа CI/CD серед публічних проєктів — більшість студентів вже мають акаунти на GitHub, тож саме її використано як основну для комп'ютерного практикуму № 3. Конфігурація живе у каталозі `.github/workflows/` у вигляді YAML-файлів.

Приклад `.github/workflows/ci.yml` для NestJS-проєкту:

```

name: CI

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

jobs:
  test:
    runs-on: ubuntu-latest
    steps:

```

```

- uses: actions/checkout@v4
- uses: actions/setup-node@v4
  with:
    node-version: 24
    cache: npm
- run: npm ci
- run: npm run lint
- run: npm test

build-image:
  runs-on: ubuntu-latest
  needs: test
  if: Github.ref == 'refs/heads/main'
  steps:
    - uses: actions/checkout@v4
    - uses: docker/login-action@v3
      with:
        registry: ghcr.io
        username: ${GITHUB_ACTOR}
        password: ${GITHUB_TOKEN}
    - uses: docker/setup-buildx-action@v3
    - uses: docker/build-push-action@v5
      with:
        push: true
        tags: ghcr.io/${GITHUB_REPOSITORY}:${GITHUB_SHA}
        cache-from: type=gha
        cache-to: type=gha,mode=max

```

Ключові концепції: триггери `on` (`push`, `pull_request`, `schedule`, `workflow_dispatch`); `jobs` виконуються паралельно за замовчуванням, з `needs` створюються залежності; `actions/*` — готові кроки з GitHub Marketplace; `secrets.GITHUB_TOKEN` — автоматичний токен для доступу до власного репозиторію та GitHub Container Registry (`ghcr.io`). Секрети додаються через `Settings → Secrets and variables → Actions`.

Зверніть увагу на `cache-from: type=gha` та `cache-to: type=gha,mode=max` — це 9.4 GitHub Actions Cache для шарів Docker. Перша збірка триває як завжди, але всі наступні (при незмінних залежностях) використовують кешовані шари — час `build-image` скорочується від кількох хвилин до 10–20 секунд. У промислових пайплайнах кешування шарів обов'язкове: без нього CI-конвеєр стає вузьким горлом і команда чекає по 10 хвилин на кожен PR.

Інші корисні дії з Marketplace: `docker/setup-buildx-action` (швидша збірка з кешуванням шарів), `aws-actions/configure-aws-credentials` (деплой у AWS), `codecov/codecov-action` (покриття тестами). Усі вони документовані на <https://github.com/marketplace?type=actions>.

9.6 Питання та завдання для самоконтролю

1. Чим Docker-образ відрізняється від Docker-контейнера?
2. Що таке Dockerfile і яка роль кожної з ключових інструкцій (FROM, WORKDIR, COPY, RUN, EXPOSE, CMD)?
3. Що таке multi-stage build і чому він зменшує розмір production-образу?
4. Для чого потрібен файл .dockerignore?
5. Що таке Docker Compose і коли він потрібен?
6. Які типи елементів описуються у docker-compose.yml (services, volumes, networks)?
7. Чим відрізняються depends_on і healthcheck?
8. Що означає принцип 12-Factor App у частині конфігурації?
9. Як організовано .env, .env.local, .env.production і чому останні два не комітяться?
10. Що таке CI-пайплайн і з яких етапів він зазвичай складається?
11. Чим 9.4 GitHub Actions відрізняється від 9.3 GitLab CI на рівні конфігурації?
12. Що таке GitHub Container Registry (ghcr.io) і як до нього публікувати образи?
13. Як зберігати секрети (паролі, API-ключі) у CI-пайплайнах?
14. Чому FTP-розгортання вважається застарілим і не використовується у промислових проєктах?

ПЕРЕЛІК ПОСИЛАНЬ

1. Технології розробки веб-ресурсів [Електронний ресурс] : навчальний посібник / В. П. Молчанов, О. К. Пандорін. – Харків : ХНЕУ ім. С. Кузнеця, 2019. – 130 с. – ISBN 978-966-676-754-0.
2. Різниця між HTTP та HTTPS. HostIQ Wiki. URL: <https://hostiq.ua/wiki/ukr/http-https/> (дата звернення: 19.05.2026).
3. Фрімен Е. Head First. Патерни проєктування. – К. : ТОВ «Фабула», 2020. – 672 с.
4. Ніколаєнко О. Ю. Використання НІТ у курсі «Технології створення Web-вузлів» // Вісник. – К. : НПУ імені М. П. Драгоманова, 2012. – Вип. 3. – С. 76–78.
5. PHP Manual : офіційна документація мови PHP. Version 8.3. The PHP Group. URL: <https://www.php.net/manual/en/> (дата звернення: 03.05.2026).
6. Мартін Р. С. Чиста архітектура : пер. з англ. – К. : ТОВ «Фабула», 2019. – 368 с.
7. Symfony Documentation : офіційний посібник з фреймворку. Version 7.x. SensioLabs. URL: <https://symfony.com/doc/current/> (дата звернення: 03.05.2026).
8. NestJS Documentation : офіційний посібник з фреймворку. Version 10.x. URL: <https://docs.nestjs.com/> (дата звернення: 03.05.2026).

ПЕРЕЛІК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Doctrine ORM Documentation : офіційна документація об'єктно-реляційного відображення. Version 3.x. URL: <https://www.doctrine-project.org/projects/orm.html> (дата звернення: 03.05.2026).
2. MySQL 8.0 Reference Manual : офіційна документація СУБД. Oracle Corporation. URL: <https://dev.mysql.com/doc/refman/8.0/en/> (дата звернення: 03.05.2026).
3. Prisma Documentation : сучасний type-safe ORM для Node.js та TypeScript. Version 5.x. Prisma Data Inc. URL: <https://www.prisma.io/docs/> (дата звернення: 03.05.2026).
4. TypeScript Handbook : офіційний посібник з мови. Microsoft. URL: <https://www.typescriptlang.org/docs/handbook/> (дата звернення: 03.05.2026).
5. Docker Documentation : офіційна документація платформи контейнеризації. Docker Inc. URL: <https://docs.docker.com/> (дата звернення: 03.05.2026).
6. 9.4 GitHub Actions Documentation : офіційна документація платформи неперервної інтеграції. GitHub Inc. URL: <https://docs.GitHub.com/en/actions> (дата звернення: 03.05.2026).
7. PHP Standards Recommendations (PSR) : стандарти спільноти PHP-FIG. URL: <https://www.php-fig.org/psr/> (дата звернення: 03.05.2026).
8. OWASP API Security Top 10 : найпоширеніші вразливості програмних інтерфейсів. The OWASP Foundation, 2023. URL: <https://owasp.org/API-Security/editions/2023/en/0x00-header/> (дата звернення: 03.05.2026).
9. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : PhD Dissertation. University of California, Irvine, 2000. URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (дата звернення: 03.05.2026).