МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему «РЕАЛІЗАЦІЯ БАГАТОРІВНЕВОЇ АРХІТЕКТУРИ ВЕБЗАСТОРСУНКУ НА ОСНОВІ SPRING BOOT»

Здобувача вищої освіти другого (бакалаврського)рівня,
IV курсу, групи КБ 21-1, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології (за галузями)»



(підпис)

Демченко Владислав Олексійович
(прізвище, ім'я, по батькові)

Науковий керівник 

(підпис)

Ніжегородцев Владислав Олександрович, к.пед.н. доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Керівник освітньо-професійної програми 

(підпис)

Омельчук Антон Анатолійович к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри комп'ютерних та інформаційних технологій і систем протокол від «13» травня 2025 № 18

Завідувач кафедри 

(підпис)

Філоненко Михайло Миколайович, к.ф.-м.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2025

Реєстрація № 53 26.05.2025 _____
(дата) (підпис відповідальної особи)

Результати перевірки до захисту 26.05.2025 р. _____
(до захисту, на доопрацювання) (дата) (підпис)

Результат захисту 4 (добре) 04.06.2025 _____
(оцінка) (дата) (підпис)

Голова комісії: _____ В.М. Франчук
(підпис)

Члени комісії: _____ М.М. Філоненко
(підпис)

_____ С.М. Грищенко
(підпис)

_____ А.А. Омельчук
(підпис)

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії, секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст кваліфікаційної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	**від 0 до 30**	26		Ніжегородцев В.О. Франчук В.М. Філоненко М.М. Грищенко С.М. Омельчук А.А. Суховецька Л.П.
2. Дотримання вимог щодо оформлення	**від 0 до 10**	8		
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	**від 0 до 5**	5		
4. Рецензент	**від 0 до 5**	4		
РАЗОМ БАЛІВ	**від 0 до 50**	43		

Оцінювання кваліфікаційної роботи науковим керівником – 41 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
84	B	4 (добре)	04.06.2025	Франчук В.М. Філоненко М.М. Грищенко С.М. Омельчук А.А. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань: 12 «Інформаційні технології»
Спеціальність: 122 «Комп'ютерні науки»
Освітня програма: «Інформаційні управляючі системи і технології (за галузями)»

Затверджую
Завідувач кафедри _____ Філоненко М.М.
«14» листопада 2024

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Демченко Владиславу Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Реалізація багаторівневої архітектури вебзастосунку на основі Spring Boot
Затверджена наказом ректора від «13» листопада 2024 № 1717.
2. Строк здачі закінченої роботи «05» травня 2025.
3. Мета роботи: розробка багаторівневого вебзастосунку для створення та управління персональними каталогами книг із можливістю додавання нотаток
Об'єкт дослідження процес проєктування і реалізації архітектурно структурованого вебзастосунку
Предмет дослідження побудова багаторівневої архітектури вебзастосунку
Орієнтовна база дослідження виступає локальне середовище розробки, що включає сучасні інструменти, фреймворки та сервіси, необхідні для створення багаторівневого вебзастосунку
4. Вихідні дані до роботи функціональні та нефункціональні вимоги до інформаційної системи
5. Перелік графічного матеріалу структурна схема архітектури вебзастосунку; діаграми взаємодії компонентів; схема взаємодії об'єктів; схеми взаємодії архітектурних об'єктів
6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)
У першому розділі розглянуто питання, пов'язані з теоретичними аспектами архітектури вебзастосунків. У другому розділі досліджено сучасні технології та інструменти, що використовуються при створенні багаторівневих вебзастосунків. Третій розділ присвячений практичній реалізації проєкту, зокрема, питання аналізу вимог, вибору стеку технологій, проєктування архітектури, реалізації функціоналу, проведення тестування та оцінки результатів роботи застосунку.

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	До 30.10.2024	До 30.10.2024
2	**І розділ** (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	27.12.2024	27.12.2024
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	17.02.2025	17.02.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	16.04.2025	16.04.2025
5	**Висновки**	01.05.2025	01.05.2025
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	09.05.2025	09.05.2025
7	Отримання відгуку керівника	27.05.2025	27.05.2025
8	Отримання зовнішньої рецензії	29.05.2025	29.05.2025

8. Дата видачі завдання «14»листопада 2024

9. Керівник кваліфікаційної роботи _____ Ніжегородцев В.О.
(підпис) (прізвище, ініціали)

10. Керівник освітньо-професійної програми _____ Омельчук А.А.
(підпис) (прізвище, ініціали)

11. Завдання до виконання отримав здобувач вищої освіти _____ Демченко В.О.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Мета роботи: розробка багаторівневого вебзастосунка для створення та управління персональними каталогами книг.

Завдання роботи:

1. Аналіз архітектурних підходів для вебзастосунків;
2. Дослідження сучасних технологій і засобів для реалізації багаторівневої архітектури;
3. Проектування структури та архітектури вебзастосунка;
4. Реалізація серверної частини (Backend);
5. Організація взаємодії з базою даних та реалізація функціоналу фільтрації й пошуку;
6. Проведення тестування та оцінка практичного застосування розробленого рішення.

Предметом дослідження є вебзастосунок для управління каталогами книг.

Базою дослідження є сучасні підходи до архітектурного проєктування, фреймворк Spring Boot, бібліотеки Java, а також приклади функціональних вебзастосунків, що дозволяють структурувати дані.

За результатами дослідження створено прототип багаторівневого вебзастосунка з можливістю додавання нотаток до книг, фільтрації та зручного пошуку.

Одержані результати можуть бути використані як база для створення персоналізованих цифрових бібліотек або сервісів для ведення особистих архівів.

Кваліфікаційна робота складається зі 46 сторінок без додатків, 8 рисунків, 39 джерел.

Ключові слова: домашній менеджер, інтерефейс, вебзастосунок.

Рік виконання кваліфікаційної роботи: 2025.

Рік захисту роботи: 2025.

ABSTRACT

Objective of the work: Development of a multi-tier web application for creating and managing personal book catalogs.
Tasks of the work are:

1. Analysis of architectural approaches for web applications
2. Research of modern technologies and tools for implementing multi-tier architecture
3. Designing the structure and architecture of the web application
4. Implementation of the server-side (Backend)
5. Organizing interaction with the database and implementing filtering and search functionality
6. Conducting testing and evaluating the practical application of the developed solution

The subject of the research is a web application for managing book catalogs.

The basis of the research is: the research is based on modern approaches to software architecture, the Spring Boot framework, Java libraries, and examples of functional web applications that allow for data structuring. According to the research results, a prototype of a multi-tier web application was developed with functionality for adding notes to books, filtering, and convenient search.

The obtained results can be used as a foundation for creating personalized digital libraries or services for managing private archives.

The qualification work consists of 46 pages without appendices, 8 figures, 39 sources.

Keywords: personal manager, interface, web application.

Year of completion of the qualification work: 2025.

Year of defense of the work: 2025.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 3

ВСТУП 4

РОЗДІЛ 1. АНАЛІЗ АРХІТЕКТУРНИХ ПІДХОДІВ ДЛЯ ВЕБЗАСТОСУНКІВ **7**

1.1. Поняття архітектури програмного забезпечення 7

1.2. Клієнт-серверна модель: принципи та варіації 10

1.3. Основні типи архітектур для вебзастосунків 13

1.4. Порівняльний аналіз архітектурних рішень 15

РОЗДІЛ 2. АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ РЕАЛІЗАЦІЇ БАГАТОРІВНЕВОЇ АРХІТЕКТУРИ ВЕБЗАСТОСУНКА 18

2.1 Архітектурні аспекти вебзастосунка 18

2.2. Бекенд розробка: Основні концепції та технології 22

2.3. Технології та інструменти зберігання даних 24

2.4. Допоміжні інструменти та бібліотеки 30

РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКА 35

3.1 Аналіз проєкту та обґрунтування технологій 35

3.2. Реалізація багаторівневої архітектури застосунку 39

3.3. Інструменти, тестування та підтримка проєкту 44

ВИСНОВКИ 45

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 47

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API — (Application Programming Interface) програмний інтерфейс додатків

CRUD — основні операції над даними: створення, читання, оновлення, видалення

HTTP — протокол передавання гіпертексту

JSON — текстовий формат обміну даними між додатками.

JVM — віртуальна машина Java для виконання Java-додатків

MVC — архітектурний шаблон

REST — (Representational State Transfer) архітектурний стиль взаємодії систем у мережі

SOAP — (Simple Object Access Protocol) протокол обміну структурованими повідомленнями

SQL — (Structured Query Language) мова структурованих запитів для роботи з базами даних

NoSQL — тип систем управління базами даних, що не базуються на реляційній моделі

ORM — (Object-Relational Mapping) спосіб взаємодії об’єктів додатка з реляційною базою даних

DAO — (Data Access Object) шаблон доступу до даних у програмній архітектурі

DTO — (Data Transfer Object) об’єкт для передачі даних між шарами застосунку

CRUD-операції — створення, читання, оновлення, видалення даних у базі.

Spring Boot — фреймворк для спрощеної розробки додатків на платформі Spring

MongoDB — документоорієнтована NoSQL база даних

MySQL — реляційна система управління базами даних

DI — це шаблон проєктування, в якому залежності впроваджуються, або передаються по посиланню

ВСТУП

Сучасний світ неможливо уявити без цифрових технологій, які все глибше проникають у повсякденне життя та професійну діяльність. У цьому контексті вебзастосунки відіграють ключову роль у забезпеченні зручного доступу до інформації, організації даних і взаємодії користувачів із системами. Особливої актуальності набуває розробка персоналізованих інструментів, які дозволяють зберігати, структурувати та ефективно знаходити потрібну інформацію.

Метою кваліфікаційної роботи є розробка багаторівневого вебзастосунку для створення та управління персональними каталогами книг із можливістю додавання нотаток.

Для досягнення поставленої мети у роботі передбачено виконати такі завдання:

— вивчити сучасні підходи до архітектури вебзастосунків та методи побудови багаторівневих систем на основі мови Java. Особлива увага приділяється аналізу принципів розподілу логіки між шарами — контролера, сервісу та доступу до даних;

— визначити вимоги до функціональності системи, яка дозволяє створювати та керувати каталогами книг. До таких вимог належить реалізація базових CRUD-операцій, можливість додавання коментарів до книг, створення нотаток, формування користувацьких добірок, а також пошук по полях назви та опису із частковим збігом;

— розробити архітектурну модель застосунку, яка забезпечує чіткий розподіл обов’язків між компонентами, сприяє легкості у підтримці та можливості розширення функціоналу;

— реалізувати систему фільтрації та контекстного пошуку, яка дозволяє користувачеві швидко знаходити потрібні книги не лише за точним збігом, але і за частковими входженнями в назві або описі;

— здійснити тестування бізнес-логіки розробленої системи. Це дозволить перевірити коректність виконання ключових функцій застосунку;

— оцінити наукову і практичну цінність створеного застосунку як готового до використання рішення для кінцевих користувачів. У межах цього

завдання передбачається розгляд можливостей масштабування застосунку, інтеграції з зовнішніми API (наприклад, Google Books);

Об'єктом дослідження виступає процес проєктування і реалізації архітектурно структурованого вебзастосунку, який дозволяє створювати та керувати персональними каталогами книг.

Предметом дослідження є побудова багаторівневої архітектури вебзастосунку з використанням Java, фреймворку Spring Boot та інших інструментів, необхідних для реалізації функціоналу і розширених можливостей взаємодії з базою даних.

Методи дослідження для реалізації поставлених завдань у роботі використано загальнонаукові методи аналізу, синтезу, моделювання та порівняння. Під час побудови архітектури застосовувалися методи структурного аналізу, проектування з урахуванням принципів SOLID та REST-архітектури.

База дослідження є досвідом застосування сучасних IT-технологій та фреймворків у сфері веброзробки. У процесі роботи використовувались локальні середовища розробки: IntelliJ IDEA, Docker, база даних PostgreSQL, а також технологічний стек Spring Boot, який широко використовується в розробці корпоративних систем.

Наукова та практична значимість одержаних результатів полягає у можливості ефективного керування особистими каталогами книг за допомогою розробленого вебзастосунка. Він дозволяє створювати каталоги, додавати нотатки та здійснювати пошук за назвами, що сприяє зручній роботі з великими обсягами персональної інформації. Система має модульну та масштабовану структуру, легко адаптується під інші типи даних і може бути інтегрована в складніші інформаційні платформи.

Особистий внесок здобувача полягав у самостійному виконанні всієї роботи, що включала аналіз предметної області, проєктування архітектури, розробку програмного забезпечення та тестування функціоналу.

Апробація результатів роботи здійснювалася під час внутрішнього семінару кафедри, де результати отримали схвальні відгуки та були рекомендовані до використання в навчальних цілях як приклад архітектурно структурованого вебзастосунку. Крім того, створений застосунок був використаний для персонального керування книжковими добірками, що підтверджує його практичну ефективність.

РОЗДІЛ 1. АНАЛІЗ АРХІТЕКТУРНИХ ПІДХОДІВ ДЛЯ ВЕБЗАСТОСУНКІВ

1.1. Поняття архітектури програмного забезпечення

Архітектура програмного забезпечення — це фундаментальна структура програмної системи, що визначає компоненти, їхні взаємозв’язки та правила взаємодії між ними. Що є основою для проєктування, розробки та подальшого супроводу програмного продукту. Чітке визначення архітектури дозволяє ефективно управляти складністю системи, забезпечує масштабованість та спрощує супровід.

Ключовими характеристиками архітектури є модульність, ієрархічність, узгодженість, відокремлення відповідальностей та незалежність від конкретних технологій реалізації. Архітектурні рішення приймаються на ранніх етапах розробки, і вони мають довготривалий вплив на ефективність, стабільність та розширюваність програмного продукту. Також важливу роль відіграє документування архітектури, що сприяє кращому розумінню системи всіма учасниками проєкту.

Дослідники також наголошують, що архітектура повинна бути здатною адаптуватися до змін у вимогах, що особливо важливо в умовах динамічного цифрового середовища [10, с. 178].

Якість програмних систем та їх відповідності функціональним і нефункціональним вимогам функціонує завдяки ретельно визначеній архітектура програмного забезпечення. Спроєктована архітектура є запорукою успішної реалізації програмного продукту та зниження ризиків на всіх етапах життєвого циклу.

Архітектура програмного забезпечення розглядається не лише як технічна структура, а і як набір стратегічних рішень, які визначають подальший розвиток продукту [4, с. 16].

Головною метою архітектури програмного забезпечення є забезпечення чіткої структуризації системи, яка дозволяє керувати її складністю, забезпечує гнучкість, масштабованість і стабільність у процесі розвитку. Архітектура

виступає як план будівництва, що описує основні компоненти системи, принципи їхньої взаємодії та технологічні обмеження, яких слід дотримуватися під час розробки [13, с. 22].

Визначення таких підходів дає змогу досягти високого рівня організованості в процесі розробки та забезпечити контроль за якістю рішень. Це також дозволяє ефективніше розподіляти ресурси, уникати дублювання функціоналу та пришвидшити розробку шляхом повторного використання перевірених архітектурних рішень.

Серед ключових завдань архітектури — визначення ролей і відповідальностей компонентів системи, створення чітких інтерфейсів для взаємодії між модулями, підтримка можливості повторного використання рішень, забезпечення гнучкості до змін та адаптації, а також дотримання нефункціональних вимог — таких як безпека, продуктивність, надійність та масштабованість [10, с. 177].

Архітектура має передбачати можливість інтеграції нових технологій та компонентів без радикального перероблення всієї системи. Крім того, якісна архітектура допомагає знизити ризики на пізніх етапах проєкту, спрощуючи процеси тестування, впровадження й обслуговування.

Чітке структурування системи дозволяє зменшити час на виявлення та усунення помилок, полегшує інтеграцію з іншими системами, тестувальників і DevOps-інженерів [8, с. 118].

Важливою задачею архітектури є також створення спільного бачення для команди розробників, аналітиків і замовника. Вона виступає як міст між бізнес-вимогами та технічною реалізацією, тому повинна бути зрозумілою та добре документованою.

Вже на етапі проектування архітектура забезпечує основу для прийняття технічних рішень, які впливають на майбутню підтримку, розширюваність і масштабованість продукту. Погано продумана архітектура часто призводить до складнощів із впровадженням нових функцій або до зростання витрат на підтримку [10, с. 179].

Водночас важливість архітектури проявляється і в тому, що вона є посередником між бізнес-цілями та технічною реалізацією. Її завдання — знайти баланс між короткостроковими вигодами та довгостроковими перспективами. У сучасній розробці особливо цінується здатність архітектури підтримувати еволюцію продукту без суттєвих змін у фундаментальних компонентах.

Архітектура програмного забезпечення безпосередньо впливає на всі інші складові процесу розробки — від управління проєктами до забезпечення якості продукту. Вона визначає загальний каркас системи, в межах якого працюють учасники команди. Саме архітектурні рішення встановлюють рамки, у яких формулюються вимоги, будуються модулі та обираються технології [12, с. 45].

Архітектура відіграє важливу роль у процесах безперервної інтеграції та доставки (CI/CD), адже саме вона визначає, наскільки ефективно можна автоматизувати побудову, тестування та розгортання програмного забезпечення. Наприклад, мікросервісна архітектура зазвичай краще підходить для CI/CD-підходів, оскільки дозволяє незалежно розгортати окремі компоненти.

Водночас, вона тісно пов’язана з вибором шаблонів проєктування, структурою команд, підходами до тестування, а також стратегіями моніторингу й логування. Архітектура системи не існує у вакуумі — її ефективність напряму залежить від взаємодії з іншими аспектами розробки, зокрема з управлінням ризиками, бюджетуванням, UX-дизайном та забезпеченням інформаційної безпеки [10, с. 180].

Зрештою, продумана архітектура програмного забезпечення — це інструмент досягнення високої якості продукту, здатного ефективно функціонувати в умовах змінних вимог і середовища.

1.2. Клієнт-серверна модель: принципи та варіації

У сучасному веброзробленні особливу увагу приділяють архітектурним підходам. Однією з найпоширеніших і найстійкіших моделей у цій сфері є клієнт-серверна архітектура. Її популярність зумовлена здатністю чітко розмежовувати функціональні обов'язки між клієнтською та серверною частинами системи, що сприяє легшій розробці, оновленню та підтримці програмного забезпечення. У цьому підрозділі розглянемо основні принципи побудови клієнт-серверної моделі, її варіації, а також переваги, які вона надає під час створення сучасних вебзастосунків.

Клієнт-серверна модель є фундаментальною архітектурною концепцією в побудові розподілених систем і вебзастосунків. Вона базується на розділенні ролей між двома головними учасниками: клієнтом, який надсилає запити, та сервером, який ці запити обробляє та повертає відповідь. Таке розмежування забезпечує структурну організацію, яка полегшує масштабування, модульність та обслуговування системи [5, с. 38].

Одним із ключових принципів є розділення обов'язків. Клієнт зосереджується на інтерфейсі користувача та взаємодії, тоді як сервер обробляє бізнес-логіку, доступ до бази даних і виконання операцій. Це дозволяє окремо змінювати або вдосконалювати кожну сторону без істотного впливу на іншу.

Ще одним важливим аспектом є модель запит–відповідь: клієнт надсилає конкретний запит (наприклад, HTTP-запит у вебзастосунках), а сервер обробляє його та повертає результат. У класичній реалізації сервер не зберігає стану клієнта між запитами, що підвищує масштабованість — така поведінка називається stateless.

У системах на базі клієнт-серверної архітектури також часто впроваджується багаторівнева безпека. Сервер зазвичай перевіряє автентичність клієнта, а також має механізми контролю доступу до ресурсів. Ці засоби безпеки критично важливі в умовах публічних мереж і великої кількості користувачів.

Архітектура клієнт-сервер добре узгоджується з принципами об'єктно-орієнтованого програмування та сервісно-орієнтованої архітектури, що дозволяє

легко масштабувати систему горизонтально, тобто шляхом додавання нових серверів або екземплярів.

Клієнт-серверна модель стала базовою для багатьох сучасних архітектурних рішень, зокрема мікросервіси. Саме завдяки її простоті й ефективності вона залишається актуальною й досі [16, с. 61].

Завдяки клієнт-серверній архітектурі, керування ресурсами стає значно ефективнішим. Централізоване зберігання та обробка даних на сервері спрощує адміністрування, оновлення, контроль доступу та підвищує рівень безпеки інформаційної системи. Саме ці переваги зробили дану модель затребуваною при розробці різноманітних інформаційних систем [5, с. 41].

Масштабованість — ще одна суттєва перевага. Систему можна масштабувати вертикально або горизонтально. Це дозволяє підтримувати велику кількість одночасних клієнтів без втрати продуктивності. Крім того, поділ обов’язків між клієнтом і сервером дозволяє ефективно використовувати ресурси, розвантажуючи клієнтську частину від складних обчислень.

Завдяки чіткому розмежуванню функціоналу, модель є зручною для розробки та супроводу. Можна розвивати клієнтські та серверні частини незалежно одна від одної, за потреби змінюючи інтерфейси, логіку або навіть мову програмування, на якій реалізовано один з компонентів.

Однак клієнт-серверна модель не позбавлена недоліків. Головним недоліком є потенційна залежність від сервера. Якщо сервер виходить з ладу або перевантажується, вся система може перестати працювати. Тому критично важливо впроваджувати механізми відмовостійкості та резервного копіювання, що збільшує загальні витрати на інфраструктуру.

Варто також зазначити, що передача даних по мережі створює ризики безпеки. Необхідно впроваджувати механізми шифрування, автентифікації та авторизації, щоб уникнути витоку інформації та атак типу "людина посередині".

Попри ці виклики, клієнт-серверна модель залишається ефективним рішенням, особливо для систем, де потрібен централізований контроль, передбачуване навантаження й модульна структура [16, с. 64].

Класична клієнт-серверна модель, що передбачає наявність одного сервера та кількох клієнтів, з часом еволюціонувала у низку варіацій. Кожна з них адаптується під специфічні потреби застосунків, масштабування, надійності та швидкості обробки даних. Найпоширенішими є двохрівнева, трирівнева та багаторівнева моделі.

Цікаво, що з розвитком технологій усе більшої популярності набуває багаторівнева (n-tier) архітектура, у якій кожна функціональна складова реалізується окремим рівнем або сервісом. Такий підхід дозволяє масштабувати кожен компонент незалежно, розподіляти навантаження та забезпечувати високу відмовостійкість системи. Наприклад, у сучасних вебзастосунках можна виокремити фронтенд, API-шлюз, мікросервіси бізнес-логіки, сервер авторизації та базу даних як окремі рівні.

Залежно від конкретного проєкту, ресурсу та вимог до безпеки й продуктивності, кожна з моделей може мати свої переваги або обмеження. Вибір відповідної варіації клієнт-серверної архітектури має ґрунтуватися на аналізі потреб системи, навантаження, типу користувачів та інфраструктурних можливостей [16, с. 46].

Клієнт-серверна модель є фундаментальною архітектурною основою більшості сучасних вебзастосунків. Її принцип роботи — забезпечити логічну організацію процесів обробки, передачі та зберігання даних. У типовому вебзастосунку клієнтська сторона — це браузер або мобільний застосунок, який взаємодіє із сервером через HTTP/HTTPS-запити.

На клієнтському боці зазвичай реалізується інтерфейс користувача та базова логіка взаємодії, що дозволяє забезпечити швидкий відгук системи. Серверна частина, виконує функції бізнес-логіки, обробки запитів, авторизації та доступу до бази даних. [5, с. 54].

Клієнт-серверна модель особливо ефективна при реалізації RESTful API та SPA де взаємодія між клієнтом і сервером відбувається у формі асинхронного обміну даними. Така структура дозволяє зменшити навантаження на сервер при повторному відображенні сторінок, підвищуючи швидкість роботи системи.

1.3. Основні типи архітектур для вебзастосунків

Починаючи з двошарової архітектури порівнюючи з іншими вона, може виявитись менш оптимальною для великих корпоративних систем, де критично важливі масштабованість, гнучкість і високий рівень безпеки. У таких випадках введення додаткового проміжного шару може покращити керованість та розділити відповідальності.

Попри певні обмеження, двошарова архітектура зберігає свою актуальність для невеликих і середніх застосунків, де важлива простота розробки та розгортання. Однак, при виборі цієї архітектури для складних проектів необхідно ретельно аналізувати всі вимоги та потенційні наслідки. [38, с. 55].

Трьохшарова архітектура спрямованим на вирішення обмежень двошарової архітектури, особливо в контексті складних корпоративних систем. Основна ідея полягає у введенні додаткового проміжного шару між клієнтом та сервером даних. Цей підхід сприяє кращому розділенню відповідальностей та підвищенню гнучкості системи.

Основними компонентами трьохшарової архітектури є рівень представлення (presentation tier), рівень застосунків (application tier, або бізнес-логіки) та рівень даних (data tier). Рівень представлення відповідає за відображення інформації користувачеві та обробку його дій.

Рівень застосунків відіграє важливу роль у забезпеченні безпеки та цілісності даних, він контролює доступ клієнтів до інформації Відокремлення рівня представлення від рівня даних підвищує масштабованість системи, оскільки кожен рівень може масштабуватися незалежно відповідно до своїх потреб. [31, с. 120]

Багаторівнева архітектура є еволюцією підходів до проєктування складних програмних систем, що передбачає їх поділ на логічні рівні, кожен з яких виконує чітко визначену функцію.

Цей підхід спрямований на досягнення кращої організації коду, полегшення розробки, тестування та підтримки. Кількість рівнів може варіюватися залежно від складності системи, але загальна мета полягає у

відокремленні різних аспектів застосунку, таких як представлення даних, бізнес-логіка та управління даними [33, с. 150].

Однією з ключових переваг багаторівневої архітектури є значне покращення масштабованості та гнучкості. Оскільки кожен рівень може масштабуватися незалежно, система здатна ефективніше обробляти зростаюче навантаження. Наприклад, якщо зростає кількість користувачів, можна збільшити кількість серверів на рівні представлення, не впливаючи на інші рівні.

Прикладами типових рівнів у багаторівневій архітектурі є рівень представлення (UI), рівень бізнес-логіки (application logic), та рівень даних (data access). Однак, складніші системи можуть включати додаткові рівні, такі як рівень сервісів, рівень інтеграції або рівень кешування. Кожен рівень взаємодіє зі сусідніми рівнями через чітко визначені інтерфейси, що забезпечує слабку зв'язаність між компонентами системи [37, с. 95].

Багаторівнева архітектура також позитивно впливає на безпеку системи. Оскільки доступ до критично важливих даних може бути обмежений певними рівнями, це ускладнює несанкціонований доступ та підвищує загальний рівень захищеності застосунку.

Чітке розділення відповідальностей між рівнями полегшує роботу різних команд розробників, кожна з яких може спеціалізуватися на певному аспекті системи, що в кінцевому підсумку призводить до підвищення якості програмного забезпечення [34, с. 78].

Мікросервісна архітектура являє собою підхід до розробки програмного забезпечення, в якому великий застосунок розбивається на набір невеликих, незалежно розгортаємих сервісів. Кожен мікросервіс виконує бізнес-функцію та взаємодіє з іншими сервісами через чітко визначені API, часто використовуючи легкі протоколи, такі як HTTP.

Завдяки своїй децентралізованій природі, мікросервісна архітектура забезпечує кращу стійкість до відмов. Якщо один мікросервіс виходить з ладу, це не обов'язково призводить до зупинки всієї системи. Інші сервіси можуть продовжувати працювати, а несправний сервіс може бути швидко відновлений. [35, с. 45]

Мікросервіси зазвичай організовані навколо бізнес-можливостей, а не технологічних шарів. Кожна команда може відповідати за один або кілька мікросервісів, використовуючи технологічний стек, який найкраще підходить для конкретної задачі. Це сприяє автономності команд та підвищує гнучкість розробки.

1.4. Порівняльний аналіз архітектурних рішень

Визначення чітких критеріїв є фундаментальним кроком у процесі порівняльного аналізу різних архітектурних рішень для програмного забезпечення. Без об'єктивних метрик та характеристик порівняння, вибір оптимальної архітектури стає суб'єктивним і може не відповідати реальним потребам проєкту. Ретельно визначені критерії дозволяють провести структуровану оцінку сильних та слабких сторін кожного розглянутого варіанту.

Одним з важливих критеріїв є масштабованість. Цей аспект відображає здатність архітектури ефективно обробляти зростаюче навантаження, будь то збільшення кількості користувачів, обсягу даних або транзакцій.

Іншим ключовим критерієм є продуктивність. Він охоплює такі характеристики, як час відгуку системи, пропускна здатність та ефективність використання ресурсів.

Не менш важливим є критерій підтримуваності. Він відображає легкість внесення змін, виправлення помилок та загальну зручність супроводу системи протягом її життєвого циклу. Архітектури зі зрозумілою структурою та чітким розділенням відповідальностей зазвичай є більш підтримуваними [39].

Крім того, слід враховувати критерій надійності. Він характеризує здатність системи функціонувати без збоїв та швидко відновлюватися після них. Архітектурні рішення з вбудованими механізмами відмовостійкості та резервування мають вищий рівень надійності. Важливим аспектом також є безпека, яка охоплює захист даних від несанкціонованого доступу, модифікації або розголошення. Різні архітектури можуть надавати різні рівні вбудованих механізмів безпеки [30, с. 85].

Вибір найбільш підходящого архітектурного рішення для програмного проєкту є критично важливим етапом, який безпосередньо впливає на успіх усього підприємства. Оптимальна архітектура повинна не лише відповідати поточним вимогам, але й враховувати майбутні потреби та обмеження проєкту. Процес вибору вимагає глибокого розуміння як функціональних, так і нефункціональних вимог. Функціональні вимоги визначають, що саме повинна робити система, які функції вона повинна надавати користувачам. Аналіз цих вимог допомагає визначити основні компоненти системи та їх взаємодію.

Процес вибору оптимальної архітектури часто включає розгляд кількох альтернативних варіантів та їх порівняльний аналіз на основі визначених критеріїв які можна представити у вигляді таблиці (табл. 1.1.)

Таблиця 1.1.

Порівняльна таблиця архітектур

Критерій	**Двошарова архітектура**	**Трьохшарова архітектура**	**Мікросервісна архітектура**
Масштабованість	Низька	Середня	Висока
Продуктивність	Висока при простій логіці	Висока при правильній реалізації	Залежить від реалізації
Підтримуваність	Низька	Середня	Висока
Надійність	Середня	Середня	Висока
Безпека	Базова	Середня	Висока
Вартість	Низька	Середня	Висока

Вибір архітектурного рішення є компромісом, який враховує всі аспекти проєкту. Не існує універсального рішення, яке підходило б для всіх випадків. Найкраща архітектура – це та, яка найкращим чином відповідає конкретним вимогам, обмеженням та цілям проєкту.

Розгляд практичних прикладів порівняння архітектурних рішень для вебзастосунків допоможе краще зрозуміти переваги та недоліки різних підходів у конкретних сценаріях.

Для невеликих вебзастосунків з низьким рівнем трафіку та простою функціональністю, таких як персональні блоги або невеликі інформаційні сайти, часто використовується монолітна архітектура. В такому випадку весь застосунок розгортається як єдине ціле. Перевагами є простота розробки та розгортання. Однак, при зростанні складності або навантаження, моноліт може стати важко масштабувати та підтримувати.

Для великих, високо навантажених вебзастосунків зі складною функціональністю, таких як соціальні мережі або платформи електронної комерції з мільйонами користувачів, мікросервісна архітектура стає більш популярним вибором. Розбиття застосунку на невеликі, незалежно розгортаємі сервіси дозволяє командам розробляти, тестувати та масштабувати окремі частини застосунку незалежно. Це забезпечує гнучкість, стійкість до відмов та можливість використання різних технологій для різних сервісів [35, с. 112].

Іншим прикладом є порівняння архітектур для вебзастосунків, що працюють з великими обсягами даних у реальному часі, таких як системи аналітики або IoT-платформи. У таких випадках можуть використовуватися архітектури, орієнтовані на реактивні архітектури, які забезпечують асинхронну обробку подій та високу пропускну здатність. Вибір конкретної архітектури залежить від специфічних вимог до обробки та аналізу даних.

Таким чином, практичні приклади показують, що вибір архітектурного рішення є контекстно-залежним і вимагає ретельного аналізу вимог конкретного вебзастосунка. Не існує універсального "найкращого" рішення, і оптимальний вибір завжди є результатом компромісу між різними факторами.

РОЗДІЛ 2. АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ ДЛЯ РЕАЛІЗАЦІЇ БАГАТОРІВНЕВОЇ АРХІТЕКТУРИ ВЕБЗАСТОСУНКА

2.1 Архітектурні аспекти вебзастосунка

REST (Representational State Transfer) — це архітектурний стиль взаємодії між клієнтом і сервером, що став стандартом для розробки вебсервісів. Основна ідея REST полягає в уніфікації інтерфейсу через використання HTTP-методів (GET, POST, PUT, DELETE тощо) для доступу до ресурсів, які представляються у вигляді URI. У контексті вебзастосунків REST надає простий, масштабований і легкий у впровадженні механізм обміну даними між різними компонентами системи.

Завдяки своїй простоті та відповідності архітектурним принципам Інтернету, REST отримав широке визнання у веброзробці. Кожен RESTful-запит до сервера є самодостатнім, що спрощує масштабування застосунку, дозволяє кешувати відповіді та забезпечує високу прозорість взаємодії. У поєднанні з форматами обміну даними, такими як JSON або XML, REST дозволяє легко інтегрувати клієнтські застосунки, мобільні сервіси, інтерфейси сторонніх API тощо [25, с. 146].

Кожен ресурс, доступний через REST API, може бути частиною окремого сервісу або мікросервісу, що дозволяє будувати розподілені системи з чітко визначеними зонами відповідальності. Наприклад, у багаторівневому застосунку окремі REST-контролери можуть обробляти запити до сервісу користувачів, замовлень чи продуктів незалежно один від одного.

Завдяки чіткій структурі та мінімальній залежності між клієнтом і сервером REST дозволяє створювати гнучкі, масштабовані та легкі у розгортанні вебзастосунки. Його підтримують практично всі сучасні вебфреймворки, включаючи Spring Boot, Express.js, Django, що робить його одним із ключових компонентів сучасної вебархітектури [9, с. 174].

Мікросервісна архітектура стала відповіддю на проблеми масштабованості, складності та повільної еволюції традиційних монолітних систем. У своїй суті вона базується на розподілі великої системи на набір дрібних, незалежних сервісів, кожен з яких виконує одну бізнес-функцію.

Завдяки цьому підходу, кожен компонент може розвиватися автономно, бути протестованим та розгорнутим без впливу на інші частини системи [25, с. 213].

Одним із основних принципів є автономність сервісів. Кожен мікросервіс повинен мати свою власну логіку, сховище даних та API для взаємодії. Це дозволяє уникати так званих тісних зв'язків, що є типовою проблемою монолітної архітектури. Автономія також дає змогу використовувати різні технології — наприклад, одна частина системи може бути реалізована на Java, інша — на Node.js або Python, що забезпечує технологічну гнучкість [35, p. 38].

Ще одним ключовим принципом є незалежне розгортання. Це означає, що зміни в одному сервісі не вимагають повторного збирання чи релізу всієї системи. Такий підхід суттєво скорочує час виходу нового функціоналу на ринок і дозволяє швидко адаптуватися до змін у бізнес-вимогах. Крім того, сервісна ізоляція знижує ризики — збій в одному мікросервісі не паралізує всю систему, як це буває в монолітах [36, p. 12].

Окрему увагу слід приділити незалежності баз даних. У мікросервісній архітектурі не допускається використання однієї спільної бази даних для всіх сервісів. Кожен мікросервіс повинен мати власне сховище даних, що дозволяє уникнути конфліктів доступу, полегшує масштабування та забезпечує кращу відповідність принципу Single Responsibility — єдиної відповідальності.

Комунікація між сервісами зазвичай відбувається через легковагові протоколи, такі як HTTP (з REST API), gRPC або асинхронні брокери повідомлень — наприклад, Apache Kafka або RabbitMQ. Асинхронність дозволяє мікросервісам працювати незалежно, підвищуючи масштабованість і відмовостійкість системи. Наприклад, якщо сервіс оплати тимчасово недоступний, запит може бути поставлений у чергу і оброблений пізніше без втрати даних [32, p. 151].

Патерни проєктування (або шаблони проєктування) є важливим інструментом для архітектури розробники програмного забезпечення, особливо у сфері вебзастосунків, де складність взаємодії між компонентами системи часто потребує чітко визначених рішень. Ці шаблони пропонують перевірені підходи

до розв’язання типових завдань, з якими стикаються розробники, зокрема в контексті архітектури, обробки запитів, керування станом та розширення функціональності [22, с. 88].

Одним з найпоширеніших шаблонів є MVC (Model-View-Controller) — патерн, що розділяє вебзастосунок на три компоненти: модель (дані та бізнес-логіка), представлення (UI) і контролер (рис. 2.1.).

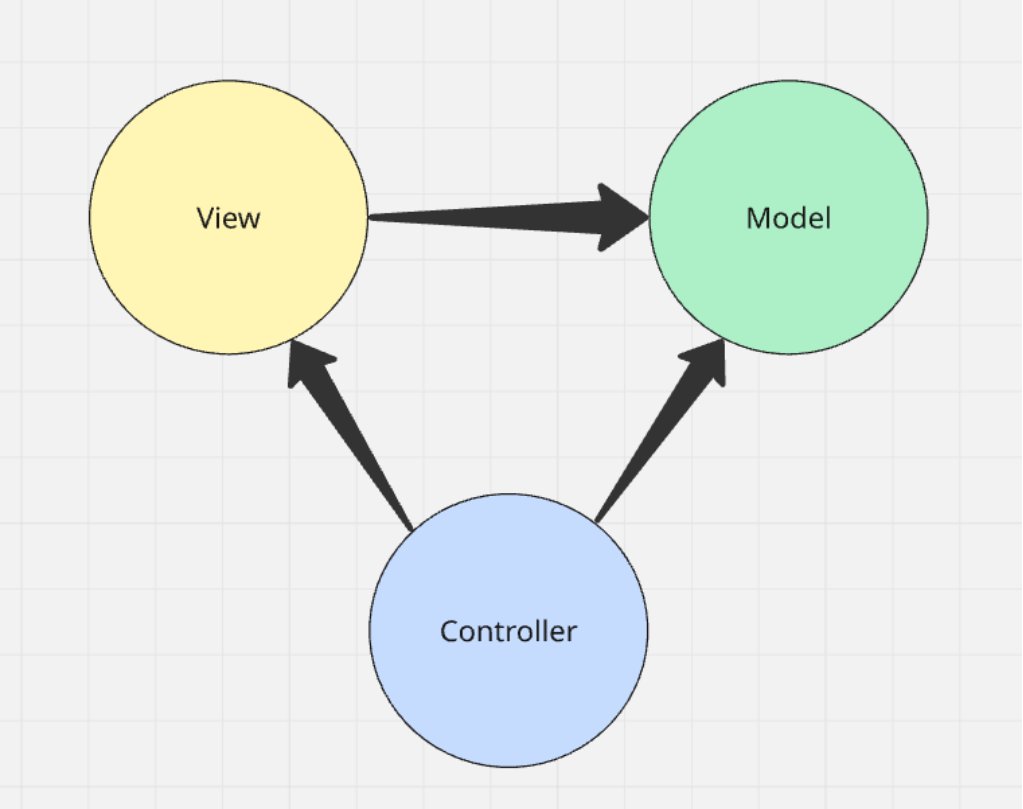


Рис. 2.1. Демонстрація патерна проєктування Model-View-Controller

Цей підхід дозволяє розробникам ізолювати різні рівні логіки, що сприяє більшій підтримуваності коду. MVC часто використовується в таких популярних фреймворках, як Spring MVC, ASP.NET MVC, Django тощо [15, с. 34].

Ще один важливий патерн — Front Controller, що передбачає наявність єдиного обробника вхідних запитів. Він відповідає за маршрутизацію запиту до відповідного контролера, виконання попередньої обробки та виклик відповідного сервісу. Цей шаблон дозволяє централізувати обробку запитів та полегшує реалізацію єдиної політики безпеки та логування. [28, с. 117].

Dependency Injection (DI) — ще один ключовий патерн, особливо важливий для сучасних вебдодатків, побудованих з використанням фреймворків. DI дозволяє зменшити залежність між компонентами, спрощує тестування, підтримку і повторне використання коду. У контексті Spring Framework, наприклад, DI є основною концепцією: контейнери керування інверсією контролю автоматично впроваджують залежності через конструктори або поля класів (рис. 2.2.).

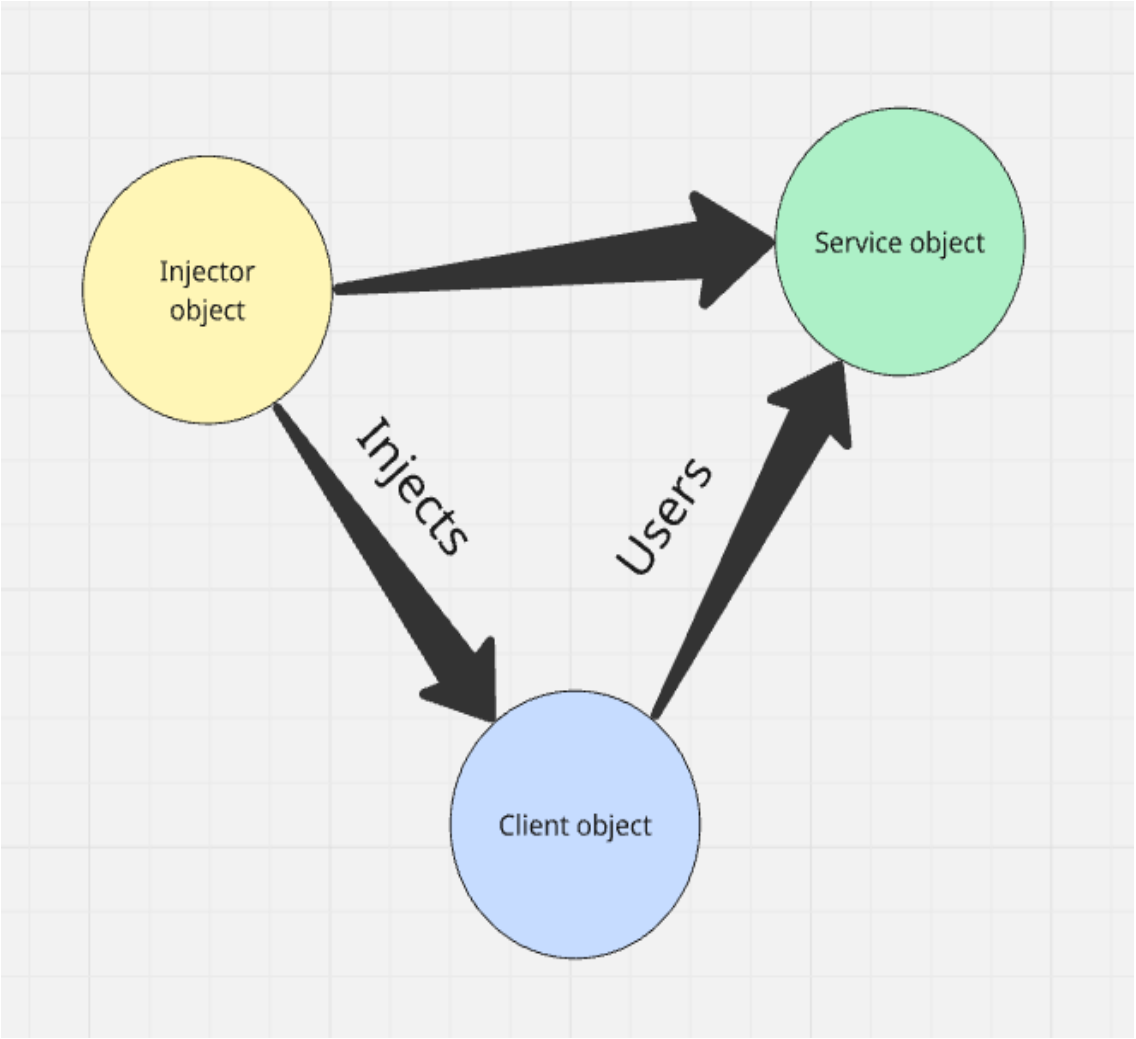


Рис.2.2. Демонстрація патерна проєктування Dependency Injection

Ці підходи забезпечують гнучкість, масштабованість і підтримуваність вебзастосунків, спрощують взаємодію та розвиток системи, а шаблони проєктування підвищують структурованість і зручність супроводу.

2.2. Бекенд розробка: Основні концепції та технології

Бекенд розробка — це частина веброзробки, яка відповідає за функціональність вебзастосунка на стороні сервера. Вона охоплює логіку обробки даних, зберігання інформації, взаємодію з базами даних, а також інтеграцію з зовнішніми сервісами. На відміну від фронтенду, який відповідає за відображення інтерфейсу, бекенд обробляє запити користувачів, керує автентифікацією, забезпечує безпеку, здійснює перевірку прав доступу, а також виконує бізнес-логіку [23, с. 11].

Найважливішим у цій сфері є розуміння, що ефективна серверна частина — це не лише про код, а й про здатність адаптуватися до змін, працювати з великими об’ємами даних і витримувати навантаження.

Ключовими завданнями бекенд-розробника є: проєктування архітектури серверної частини, створення API для взаємодії з клієнтською частиною, підключення баз даних, реалізація логіки обробки запитів і відповіді, обробка помилок, кешування та оптимізація продуктивності системи. Крім того, важливою складовою є забезпечення масштабованості, надійності та безпеки — наприклад, через захист від SQL-ін’єкцій або XSS-атак, використання HTTPS, обмеження швидкості запитів тощо [18, с. 57].

У практиці бекенд-розробки застосовуються різні інструменти залежно від мови програмування, структури проєкту та поставлених завдань. Вибір технологій безпосередньо впливає на ефективність розробки, її підтримку й майбутнє масштабування. Саме тому до підбору стеку слід підходити з урахуванням архітектурних вимог.

Java вже протягом багатьох років залишається однією з найпопулярніших та затребуваних мов програмування для розробки бекенду вебзастосунків та корпоративних систем. Її зрілість, велика екосистема та кросплатформність є значними перевагами, які приваблюють розробників та бізнес. Однією з ключових переваг Java є її незалежність від платформи, що досягається завдяки віртуальній машині Java (JVM). Код, написаний на Java, може бути запущений на будь-якій операційній системі, для якої існує JVM [24, с. 35].

Розробники використовуються різноманітні бекенд java фреймворки, і в навчальних матеріалах часто розглядаються основи роботи з популярними Java-фреймворками [20, с. 180].

Велика та активна спільнота Java розробників є ще однією вагомою перевагою. Це забезпечує велику кількість доступних бібліотек, фреймворків та інструментів, що значно спрощує та прискорює процес розробки.

Проте, Java має і певні недоліки. Одним з них є відносно висока витрата пам'яті та продуктивності порівняно з деякими іншими мовами, такими як C++. JVM потребує певних ресурсів для своєї роботи, що може бути критичним для високонавантажених систем або систем з обмеженими ресурсами. Також, багатослівність синтаксису Java може призводити до більшого обсягу коду порівняно з деякими сучасними мовами програмування [6, с. 58].

Незважаючи на ці недоліки, Java продовжує залишатися потужним інструментом для розробки бекенду завдяки своїй надійності, масштабованості та великій кількості кваліфікованих розробників на ринку праці України. Її зріла екосистема та постійний розвиток роблять її привабливим вибором для широкого спектра проєктів.

JVM пропонують функції, такі як профілювання продуктивності та інструменти моніторингу, які можуть бути корисними для діагностики та оптимізації бекенд застосунків. Вибір JVM також може залежати від операційної системи та апаратної архітектури сервера, оскільки деякі JVM можуть бути краще оптимізовані для певних платформ [24, с. 72].

Підхід JVM до компіляції коду JIT(Just-In-Time), коли компілятори аналізують найбільш часто виконуваний байт-код та компілюють його в нативний машинний код для підвищення продуктивності. Різні JVM можуть використовувати різні стратегії JIT-компіляції, що може впливати на час запуску застосунку та його пікову продуктивність.

Варто зазначити, що досі активно досліджуються питання оптимізації Java-застосунків, включаючи налаштування JVM для досягнення максимальної продуктивності в умовах високого навантаження [7, с. 95].

Завдяки бібліотекам та фреймворкам, які надаючи готові рішення для типових завдань, таких як маршрутизація запитів, обробка даних, взаємодія з базами даних, забезпечення безпеки та багато іншого, що є значущими перевагами розробникам - зосередитися на бізнес-логіці застосунку, а не на вирішенні низькорівневих технічних деталей.

Українські фахівці також відзначають переваги Java в контексті розробки високонавантажених систем завдяки її можливостям багатопотоковості та паралельного програмування, що дозволяє створювати ефективні та масштабовані бекенд рішення [21, с. 165].

Одним з найпопулярніших фреймворків для бекенд розробки на Java є Spring. Це потужний та гнучкий фреймворк, який надає широкий спектр можливостей, включаючи керування залежностями (Dependency Injection), аспектно-орієнтоване програмування (AOP), підтримку транзакцій, інтеграцію з базами даних та засоби для розробки RESTful API. Spring Boot, в свою чергу, спрощує початкове налаштування та розгортання Spring-застосунків.

Крім Java та Python, існує безліч інших фреймворків для розробки бекенду на різних мовах програмування, таких як Node.js (Express.js, NestJS), Ruby on Rails (Ruby), Laravel (PHP) та ASP.NET Core (.NET). Вибір конкретного фреймворку залежить від технологічного стека проєкту, вимог до продуктивності та масштабованості, а також від досвіду команди розробників.

2.3. Технології та інструменти зберігання даних

Реляційні бази даних є одним з найбільш поширених та зрілих підходів до організації та зберігання структурованих даних. В їх основі лежить реляційна модель, запропонована Едгаром Коддом у 1970-х роках, яка представляє дані у вигляді набору взаємопов'язаних таблиць. Кожна таблиця складається з рядків (записів) та стовпців (полів), де стовпці визначають атрибути даних, а рядки містять конкретні значення цих атрибутів для окремих сутностей.

Однією з ключових особливостей реляційних баз даних є використання структурованої мови запитів SQL (Structured Query Language) для

маніпулювання даними. SQL дозволяє виконувати різноманітні операції, такі як вибірка, вставка, оновлення та видалення даних, а також складні запити для аналізу та отримання необхідної інформації.

Реляційні бази даних характеризуються високим рівнем цілісності даних завдяки підтримці обмежень цілісності, таких як первинні та зовнішні ключі, унікальні індекси та правила валідації. Первинні ключі забезпечують унікальність кожного запису в таблиці, а зовнішні ключі встановлюють зв'язки між різними таблицями, підтримуючи консистентність даних у всій базі.

Не менш важливою особливістю є підтримка ACID-транзакцій (Atomicity, Consistency, Isolation, Durability), які гарантують надійність виконання операцій над даними. Атомарність забезпечує виконання транзакції як єдиної неподільної одиниці, консистентність гарантує, що транзакція переводить базу даних з одного дійсного стану в інший, ізоляція забезпечує незалежність одночасних транзакцій, а довговічність гарантує збереження змін після успішного завершення транзакції [11, с. 90].

Для деяких архітектурних рішень реляційних баз даних є не найкращим варіантом, саме для цього існує NoSQL (Not Only SQL) база даних, яка відрізняються від традиційних реляційних баз даних у своїй моделі даних, схемі, масштабованості та підході до зберігання та обробки інформації. Вони розроблені для вирішення проблем масштабованості та гнучкості, з якими часто стикаються сучасні вебзастосунки та великі обсяги даних. MongoDB є однією з найпопулярніших NoSQL баз даних, що належить до класу документо-орієнтованих баз даних.

Однією з ключових характеристик MongoDB є використання JSON-подібних документів для зберігання даних. Ці документи є гнучкими та можуть мати різну структуру в межах однієї колекції (аналог таблиці в реляційних базах даних). Відсутність жорсткої схеми дозволяє розробникам швидше ітеративно розробляти застосунки та легше адаптуватися до змін у вимогах до даних.

MongoDB забезпечує високу горизонтальну масштабованість завдяки своїй архітектурі, яка підтримує шардування (sharding) та реплікацію.

Шардування дозволяє розподіляти дані між кількома серверами, що підвищує пропускну здатність та обсяг зберігання. Реплікація створює кілька копій даних на різних серверах, забезпечуючи відмовостійкість та доступність даних у випадку відмови одного із серверів.

Випадки використання MongoDB є різноманітними. Вона часто застосовується для вебзастосунків, мобільних застосунків, систем керування контентом (CMS), зберігання даних профілів користувачів, каталогізації продуктів та аналітики великих даних. Її гнучка модель даних добре підходить для випадків, коли структура даних може часто змінюватися або коли дані є напівструктурованими.

До NoSQL баз даних, і зокрема MongoDB ключовими вимогами є висока масштабованість та гнучкість. Їхня документо-орієнтована модель дозволяє ефективно працювати з великими обсягами даних, які можуть мати мінливу або напівструктуровану природу. Відсутність жорсткої схеми спрощує процес розробки та адаптацію до нових вимог, а можливості горизонтального масштабування забезпечують стабільну роботу застосунку при зростанні навантаження [14, с. 155].

Бази даних NoSQL можуть мати певні компроміси щодо ACID-транзакцій порівняно з реляційними базами даних, хоча MongoDB забезпечує підтримку атомарних операцій на рівні документа та транзакцій у версіях після 4.0. Вибір між реляційною та NoSQL базою даних залежить від конкретних вимог проєкту до структури даних, масштабованості, консистентності та складності запитів.

PostgreSQL та MongoDB є двома популярними системами керування базами даних, які, проте, належать до різних класів і мають різні характеристики, що визначають їхню придатність для конкретних сценаріїв використання. PostgreSQL є потужною реляційною базою даних з відкритим вихідним кодом, яка відома своєю надійністю, підтримкою ACID-транзакцій та широким набором функцій, включаючи складні SQL-запити, тригери, збережені процедури та підтримку різних типів даних, включаючи JSON.

MongoDB, навпаки, є документо-орієнтованою NoSQL базою даних, яка зберігає дані у гнучких JSON-подібних документах. Її ключовими перевагами є висока горизонтальна масштабованість та гнучкість схеми, що дозволяє швидше адаптуватися до змін у вимогах до даних. MongoDB добре підходить для випадків, коли структура даних є еволюційною або коли потрібна обробка великих обсягів неструктурованих або напівструктурованих даних.

Одним з основних критеріїв вибору між PostgreSQL та MongoDB є потреба в ACID-транзакціях та цілісності даних. Якщо для вашого застосунку критично важлива гарантія консистентності даних у складних транзакціях, PostgreSQL з її повною підтримкою ACID буде кращим вибором. MongoDB, хоча й підтримує атомарні операції на рівні документа та транзакції у пізніших версіях, може мати певні компроміси в складних багатодокументних транзакціях.

Іншим важливим критерієм є структура даних та необхідність гнучкості схеми. Якщо дані мають чітку та стабільну структуру, і планується використовувати складні SQL-запити зі зв'язками між таблицями, PostgreSQL буде більш природним вибором. Якщо ж структура даних може часто змінюватися або є нереляційною за своєю природою, гнучка схема MongoDB може значно спростити розробку та масштабування.

Масштабованість також є важливим фактором. MongoDB зазвичай простіше масштабувати горизонтально завдяки вбудованій підтримці шардування та реплікації. PostgreSQL також підтримує масштабування, але його налаштування може бути складнішим у деяких сценаріях високого навантаження.

Порівнюючи PostgreSQL та MongoDB, часто акцентується увага на їхніх фундаментальних відмінностях у моделі даних: структуровані таблиці з чітко визначеними схемами для PostgreSQL проти гнучких документо-орієнтованих структур у MongoDB. Розглядаються також різні підходи до забезпечення цілісності даних, де PostgreSQL робить ставку на строгі ACID-транзакції та реляційні зв'язки, а MongoDB пропонує атомарність операцій на рівні документа та можливості для горизонтального масштабування, що є ключовим для

високопродуктивних застосунків. Окремо вивчаються сфери застосування кожної з систем: PostgreSQL як надійний вибір для складних транзакційних систем, фінансових застосунків та випадків, де важлива консистентність даних, а MongoDB як оптимальне рішення для веброзробки, систем керування контентом та проєктів, що потребують гнучкості схеми та швидкої ітерації [11, с. 135].

Вибір між PostgreSQL та MongoDB залежить від конкретних вимог вашого проєкту, включаючи складність даних, необхідність консистентності та атомарності, вимоги до масштабованості. Оцінки критеріїв можна представити у вигляді таблиць (табл. 2.1.)

Таблиця 2.1.

Порівняння PostgreSQL та MongoDB за основними критеріями вибору

Критерій	**PostgreSQL**	**MongoDB**
Модель даних	Реляційна	Документо-орієнтована
Схема даних	Жорстка з чітко заданими типами полів	Динамічна
Масштабованість	Вертикальна	Висока горизонтальна
Підтримка транзакцій	Повна підтримка ACID	Атомарність
Цілісність даних	Висока	Обмежена
Продуктивність на великих даних	Добре підходить для структурованих обчислень	Вища продуктивність при роботі з великим обсягом даних
Гнучкість при розробці	Низька	Висока
Тип запитів	SQL-запити	Агрегація через Aggregation Framework

Безпосередньо для роботи з реляційними базами даних, як PostgreSQL для Java-застосунків існує ряд інструментів та технологій, серед яких особливе місце займає Spring Data JPA.

Spring Data JPA є потужним модулем фреймворку Spring, який значно спрощує процес взаємодії Java-застосунків з реляційними базами даних за допомогою технології JPA (Java Persistence API). JPA є специфікацією Java EE, яка визначає стандартизований спосіб відображення об'єктів Java на таблиці реляційної бази даних (Object-Relational Mapping, ORM) та надає API для виконання операцій над цими об'єктами [6, с. 155].

Spring Data JPA розширює можливості JPA, надаючи розробникам високорівневі абстракції, які зменшують обсяг шаблонного коду, необхідного для виконання типових операцій з базами даних. Однією з ключових особливостей Spring Data JPA є репозиторії (repositories). Розробнику достатньо оголосити інтерфейс репозиторію, який успадковується від спеціальних інтерфейсів Spring Data, і Spring автоматично генерує необхідну імплементацію для виконання таких операцій, як збереження, видалення, пошук та оновлення сутностей (entities) – об'єктів Java, відображених на таблиці бази даних [2, с. 180].

Spring Data JPA також підтримує автоматичну генерацію запитів на основі назв методів у репозиторіях. Наприклад, оголошення методу findByFirstName(String firstName) автоматично призведе до створення SQL-запиту для пошуку записів у відповідній таблиці, де значення стовпця, що відповідає полю firstName сутності, дорівнює переданому параметру. Це значно спрощує написання запитів для типових сценаріїв.

Крім того, Spring Data JPA надає можливості для виконання складніших запитів за допомогою анотації @Query, яка дозволяє безпосередньо визначати JPQL (Java Persistence Query Language) або нативні SQL-запити. Також підтримуються пагінація, сортування та інші корисні функції для роботи з даними.

2.4. Допоміжні інструменти та бібліотеки

У процесі розробки програмного забезпечення, незалежно від мови програмування чи архітектури проєкту, важливу роль відіграють різноманітні допоміжні інструменти та бібліотеки. Вони значно спрощують виконання рутинних завдань, підвищують продуктивність розробників, покращують якість коду та забезпечують інтеграцію з іншими системами.

У Java-середовищі склалася досить стійка екосистема інструментів, яка зарекомендувала себе на практиці. Maven є одним із найпоширеніших засобів для керування проєктом, його залежностями та життєвим циклом збірки. Він дозволяє налаштувати однотипний підхід до побудови проєктів і забезпечує просту інтеграцію з великою кількістю плагінів. Альтернативою, яка поступово набирає популярності, є Gradle — більш гнучкий і сучасний варіант, що дозволяє налаштовувати складні сценарії збірки на основі Groovy або Kotlin DSL [29, с. 93].

Кажучи саме про Apache Maven, то він є інструментом однією з найпопулярніших систем керування проєктами в екосистемі Java. Основне її призначення — автоматизація процесу збирання, тестування та доставки програмного забезпечення. Maven надає структуру для проєктів будь-якого розміру та складності, використовуючи концепцію декларативної конфігурації через XML-файл pom.xml, що дозволяє уникнути дублювання коду та зменшити помилки при налаштуванні.

Серед головних переваг Maven — централізоване керування залежностями. Розробник вказує лише назву, версію та групу потрібної бібліотеки, а система автоматично завантажує її з центрального репозиторію. (рис. 2.3.)

Крім того, Maven також автоматично підтягує транзитивні залежності, тобто ті, які потрібні бібліотекам, що використовуються безпосередньо. Це особливо зручно у великих проєктах із великою кількістю сторонніх модулів [17, с. 88].

```
<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot</artifactId>
        <version>3.4.5</version>
    </dependency>

</dependencies>
```

Рис. 2.3. Демонстрація впровадження залежності за допомогою Maven

Maven підтримує концепцію «життєвого циклу проєкту», який складається з етапів — clean, validate, compile, test, package, verify, install, deploy. Кожен етап виконується в суворій послідовності, і розробник може впроваджувати додаткову логіку за допомогою плагінів. Наприклад, можна підключити плагін для генерації JavaDoc, для запуску юніт-тестів або для побудови Docker-образу застосунку. Така модульність робить Maven не лише інструментом для компіляції, а й повноцінною платформою для супроводу життєвого циклу ПЗ.

Ще однією важливою рисою є стандартизація структури каталогів. Maven нав’язує певну структуру (src/main/java, src/test/java тощо), що забезпечує однаковість між проєктами та спрощує їх підтримку, особливо в командах або при зміні розробників. Зрозуміла структура проєкту зменшує поріг входження для нових учасників команди [26, с. 53].

Попри наявність альтернатив, таких як Gradle або Ant, Maven залишається вибором номер один для багатьох enterprise-команд. Це зумовлено його зрілістю, стабільністю, широкою документацією та підтримкою більшістю середовищ розробки, зокрема IntelliJ IDEA, Eclipse та NetBeans. Значна кількість плагінів, які підтримуються спільнотою, дозволяє розширювати функціонал без значних зусиль.

Попри деякі обмеження у гнучкості, особливо порівняно з Gradle, Maven чудово підходить для більшості типових задач Java-розробника. Крім того, саме він найчастіше використовується як основна система збірки у фреймворках, таких як Spring Boot, що робить його практично стандартом у відповідних проєктах [17, с. 91].

Одним з найвідоміших фреймворків для створення бекенд-частини вебзастосунків є Spring Boot. Його головною перевагою є автоконфігурація та можливість створення повноцінного REST-сервісу з мінімальною кількістю налаштувань. Завдяки розширенню Spring Security легко реалізувати механізми авторизації та автентифікації, а модуль Spring Data значно спрощує взаємодію з базами даних. Розробка з використанням цих інструментів пришвидшується, а код стає більш читабельним і підтримуваним.

Окрему нішу займають бібліотеки, які усувають необхідність написання шаблонного коду. Зокрема, Lombok дає змогу автоматично генерувати стандартні методи за допомогою анотацій, що значно зменшує обсяг коду. Це важливо для команд, де велика кількість однакових за структурою класів ускладнює читання та супровід проєкту [17, с. 71].

Зневаження інструментів скорочення коду знижує читабельність коду, створює більше місця для помилок та ускладнює супровід.

Lombok — це Java-бібліотека, яка інтегрується в процесі компіляції, використовуючи анотації для автоматичної генерації коду, завдяки чому згенерований код є повноцінною частиною бінарного результату, хоч у вихідному коді його немає. Найчастіше використовуваними є анотації @Getter, @Setter, @ToString, @EqualsAndHashCode, @NoArgsConstructor, @AllArgsConstructor і особливо @Data, яка об'єднує одразу декілька з них [3, с. 142].

Перевага використання Lombok полягає у значному скороченні коду та підвищенні його чистоти. Клас, який у звичайних умовах містив би десятки рядків для геттерів і конструкторів, може бути зведений до кількох рядків з відповідними анотаціями. Це дозволяє зосередитись на бізнес-логіці, а не на рутинній реалізації методів. До того ж, менша кількість коду означає менше помилок та простіше тестування.

Однак, застосування Lombok має і певні недоліки. Головним із них є зниження прозорості коду — новачкам або стороннім розробникам може бути складніше зрозуміти, що саме відбувається, оскільки частина логіки прихована на рівні компіляції. Також деякі IDE або інструменти аналізу коду можуть мати

проблеми з коректним розпізнаванням згенерованих конструкцій, хоча сучасні середовища, як-от IntelliJ IDEA, вже мають повну підтримку цієї бібліотеки.

Lombok широко використовується в Java-екосистемі, зокрема в проєктах на Spring Boot. Його популярність зумовлена тим, що він ідеально вписується у стиль розробки, орієнтований на простоту та мінімалізм. У сервісах, де кількість сутностей може сягати сотень, Lombok дозволяє значно пришвидшити написання коду та уникнути непотрібної повторюваності [27, с. 88].

Важливо зазначити, що Lombok не змінює сутність Java як строго типізованої мови, а лише оптимізує роботу з її синтаксисом. У багатьох open-source проєктах його вже використовують як стандарт де-факто для моделювання сутностей та DTO.

Будь-який застосунок потрібно тестувати задля усунення можливих проблем, в цій сфері активно використовуються JUnit — відповідно для юніт-тестів. Його застосування дає змогу налагодити гнучке тестове середовище ще до інтеграції всієї системи. Додаткові переваги надає Testcontainers — інструмент для запуску оточення з базами даних або іншими сервісами в Docker-контейнерах безпосередньо під час тестування. Це дозволяє писати тести, максимально наближені до продуктивного середовища [29, с. 112].

Особливо важливим інструментом в арсеналі юніт-тестування є бібліотека Mockito, яка надає розширені можливості для імітації поведінки залежностей об'єкта, що підлягає тестуванню. Замість того, щоб використовувати реальні екземпляри залежних класів, які можуть бути складними в налаштуванні, повільними або мати небажані побічні ефекти, Mockito дозволяє створювати їхні контрольовані замінники – так звані "мок"-об'єкти.

Mockito — це бібліотека для Java, яка дозволяє створювати mock-об'єкти, що поводяться так, як зазначено в сценарії тесту. Mock-об'єкти корисні тоді, коли реальні залежності класу є занадто складними, повільними або нестабільними для прямого використання під час тестування. Наприклад, якщо клас взаємодіє з базою даних чи зовнішнім API, його краще тестувати з використанням моків, щоб уникнути непередбачуваності [19, с. 101].

Бібліотека дозволяє задавати поведінку моків за допомогою зручного синтаксису: метод when(...).thenReturn(...) описує, як повинен поводитися об'єкт у відповідь на виклик певного методу. Крім того, Mockito підтримує валідацію викликів методів за допомогою методу verify(...), що дає змогу перевірити, чи певна дія була виконана протягом тесту. Наприклад, можна переконатися, що метод збереження в базу викликався лише один раз.

Перевага Mockito полягає у простоті його використання: розробники не витрачають час на створення вручну фейкових класів або інтерфейсів. Бібліотека автоматично генерує необхідні об'єкти під час виконання. Водночас вона добре інтегрується з іншими фреймворками для тестування, такими як JUnit чи TestNG, і є де-факто стандартом у Java-розробці [1, с. 119].

Проте слід мати на увазі, що надмірне використання моків може спотворити логіку тестування. Наприклад, при тестуванні інтеграції між компонентами не варто повністю замінювати реальні об'єкти на моки — це призведе до втрати перевірки взаємодії між модулями. У таких випадках краще використовувати spy-об'єкти або часткове мокування, яке Mockito також підтримує.

Mockito особливо цінний у проєктах, побудованих за принципами інверсії контролю та залежностей (IoC/DI), де залежності легко підміняються у тестовому середовищі. У рамках Spring Boot, наприклад, досить часто використовується анотація @MockBean, яка інтегрує Mockito з Spring-контейнером, що дозволяє створювати повноцінні тести контролерів або сервісів, не піднімаючи базу даних чи зовнішні API.

РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКА

3.1 Аналіз проєкту та обґрунтування технологій

Розроблений програмний продукт є серверною частиною вебзастосунка, що виконує функції управління каталогом книг. Основна мета полягає у забезпеченні структурованого доступу до інформації про книги, з можливістю їх збереження, перегляду, редагування та видалення. Застосунок орієнтований передусім на бекенд-логіку, надаючи REST API, яке в майбутньому може бути інтегроване з будь-яким фронтенд-рішенням — від вебінтерфейсу до мобільного застосунку. У своєму поточному вигляді проєкт може бути корисним для невеликих книжкових архівів, бібліотек, онлайн-каталогів або як навчальна база для ознайомлення з основами побудови RESTful сервісів на Java.

Програмне забезпечення підтримує базові операції над сутністю книги (Book), зокрема створення нової книги, редагування наявної, перегляд списку всіх книг або однієї за ID, а також видалення запису. Уся взаємодія здійснюється за допомогою HTTP-запитів відповідно до принципів REST. Передбачено обробку даних у форматі JSON, що дозволяє легко інтегрувати сервіс у сучасні клієнтські інтерфейси.

Архітектура застосунку побудована відповідно до принципів розділення відповідальностей — кожен рівень проєкту (контролери, сервісна логіка, репозиторії та модель предметної області) реалізує чітко визначену функцію. На рівні доступу до даних використано Spring Data JPA, що дозволяє мінімізувати кількість вручну написаного SQL-коду і зручно взаємодіяти з базою даних. Уся логіка обробки даних зосереджена в сервісному шарі, що забезпечує гнучкість у майбутній розширюваності проєкту. У контролерах реалізовано REST API, яке є точкою входу для клієнта.

Очікуваним результатом реалізації є модульний, легко супроводжуваний та розширюваний застосунок, який може слугувати основою для подальшого розвитку або використання у навчальних цілях. Структура проєкту дає змогу легко адаптувати застосунок під нові домени — наприклад, каталог фільмів, продуктів чи музичних альбомів. У перспективі система може бути доповнена

механізмами аутентифікації, авторизації, а також клієнтською частиною для інтерактивної взаємодії з користувачем.

При виборі стеку технологій для реалізації бекенд-застосунку було враховано кілька ключових факторів: популярність та підтримка інструментів, стабільність у розробці, відповідність завданням CRUD-застосунку, зручність роботи з базами даних та тестуванням. Основною мовою програмування обрано Java, оскільки вона добре підходить для побудови серверної логіки, забезпечує високу продуктивність, типобезпечність і має розвинену екосистему бібліотек та фреймворків.

Для покращення зручності розробки було використано бібліотеку Lombok, що дозволяє значно зменшити кількість шаблонного коду, як-от гетери, сетери, конструктори. Крім того, у проєкті застосовано бібліотеку Mockito, що дає змогу реалізувати юніт-тестування сервісного шару, забезпечуючи стабільність та передбачуваність функціональності. Система збірки Maven дає змогу керувати залежностями, конфігурацією й запуском застосунку, а також забезпечує стандартизацію проєктної структури.

Як основний фреймворк для розробки було використано Spring Boot, що значно спрощує створення та налаштування бекенд-застосунків. Він автоматизує багато процесів конфігурації та дозволяє зосередитись безпосередньо на бізнес-логіці. Його модульна структура дає змогу легко інтегрувати додаткові компоненти, такі як Spring Data JPA, Spring Web та Spring Validation, які також активно використовуються в цьому проєкті.

Для взаємодії з базою даних було обрано Spring Data JPA у зв’язці з Hibernate. Це рішення забезпечує зручну обробку персистентних даних без необхідності писати великі обсяги SQL-коду вручну. Замість цього використовується репозиторійний підхід з автоматично реалізованими методами доступу до бази. Таким чином, розробка значно прискорюється, а ризики виникнення помилок — зменшуються.

Для тестування було обрано бібліотеку Mockito, яка дозволяє створювати фіктивні (mock) об'єкти та перевіряти взаємодію компонентів між собою в

ізоляції. Це забезпечує гнучке та контрольоване юніт-тестування логіки застосунку.

В якості інструменту для автоматизації складання проєкту застосовано Maven. Цей інструмент не лише керує зовнішніми залежностями, але й забезпечує стандартизацію структури проєкту, дозволяє налаштувати процеси компіляції, тестування та пакування застосунку.

Проєкт реалізовано у вигляді Maven-застосунку з типовою ієрархією директорій. Основний код розміщено в каталозі src/main/java, а ресурсні файли — у src/main/resources. У межах реалізації цей інструмент виконує ключову роль у підтримці структури проєкту, автоматизації процесу компіляції, запуску тестів, формування виконуваного файлу та контролю версій залежностей.

Конфігурація проєкту зосереджена у файлі pom.xml, який містить основні метадані, набір сторонніх бібліотек, що використовуються у проєкті, а також інструкції для запуску та збирання застосунку. Maven автоматично завантажує необхідні залежності з центрального репозиторію, що дозволяє розробнику уникнути ручного пошуку та додавання бібліотек. Серед підключених бібліотек — модулі Spring Boot (Web, Data JPA, Validation), бібліотеки для тестування (JUnit, Mockito), драйвер для роботи з базою даних, а також плагіни, необхідні для компіляції та збирання виконуваного .jar-файлу.

Життєвий цикл збірки в проєкті відбувається через виконання команди mvn clean install, яка автоматично виконує послідовні фази: очищення попередніх результатів збірки, компіляцію коду, запуск тестів, збирання результату у вигляді .jar. Це дозволяє не лише забезпечити цілісність коду на кожному етапі, а й значно зменшує ризики виникнення помилок під час розгортання. Окрім того, підтримується запуск застосунку безпосередньо з Maven за допомогою команди mvn spring-boot:run, що зручно на етапі розробки.

Таким чином, Maven виступає надійною основою для управління життєвим циклом BookCatalog-проєктом, забезпечуючи стабільність, відтворюваність та зручність розробницького процесу.

Тестові класи знаходяться у src/test/java. Така структура забезпечує чітке відокремлення основної логіки від тестів та конфігурацій, що відповідає стандартам Java-розробки.

Для забезпечення зручності в навігації, підтримці та масштабуванні застосунку проєкт було розбито на логічні пакети відповідно до функціонального призначення компонентів (рис. 3.1.).

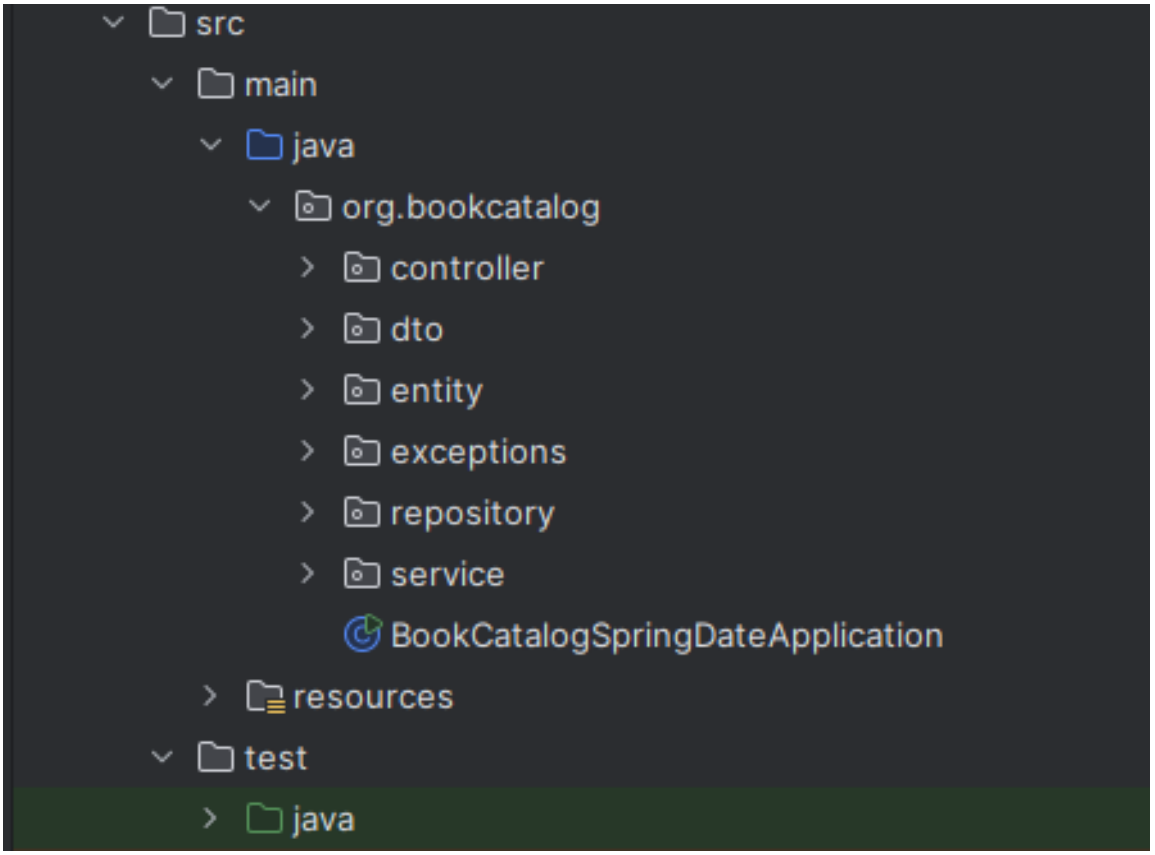


Рис. 3.1. Зображено налаштування пакетів в проєкт

Опис основних пакетів:

- com.bookcatalog.entity— містить класи предметної області (сутності), що відображають таблиці бази даних, з відповідними анотаціями JPA.

- com.bookcatalog.repository — включає інтерфейси для роботи з базою даних, що розширюють JpaRepository, забезпечуючи стандартні CRUD-операції без необхідності ручного SQL.

- com.bookcatalog.service — містить бізнес-логіку застосунку, яка інкапсулює обробку даних між рівнем доступу до БД і REST-контролерами.

- com.bookcatalog.controller — реалізує REST API, яке відповідає за обробку HTTP-запитів і повернення відповідей клієнту.
- com.bookcatalog.dto — включає класи для передачі даних між клієнтом і сервером (Data Transfer Objects), що допомагає розмежувати моделі БД від представлення.
- com.bookcatalog.exceptions — реалізовано класи для обробки виняткових ситуацій, а також глобальний обробник помилок.

Такий поділ дозволяє дотримуватись принципу розділення відповідальностей (Separation of Concerns), покращує читабельність коду та спрощує підтримку в майбутньому. Структура проєкту створює фундамент для легкої інтеграції нових функцій без порушення поточної логіки.

3.2. Реалізація багаторівневої архітектури застосунку

Предметна область застосунку моделює просту структуру книжкового каталогу, в якій основними сутностями є Book, Catalog та Note. Для реалізації цих моделей було використано звичні підходи об'єктно-орієнтованого програмування з урахуванням вимог до збереження в базі даних через Spring Data JPA.

Кожен клас-сутність розміщений у пакеті entity, що сприяє чіткому поділу відповідальностей між логічними шарами проєкту. Класи мають анотації @Entity, що дозволяє JPA автоматично відображати їх у відповідні таблиці бази даних. Поля класів відображаються на стовпці таблиць, а ідентифікатори мають анотацію @Id та автоматичну генерацію @GeneratedValue.

- Клас Catalog представляє каталоги книг, що дозволяє згрупувати книги за назвою каталогу, що слугує тематикою каталогу книг. Включає прості атрибути, такі як назва, опис, дата створення каталога, зв’язок із книгами в каталозі і може бути розширений у майбутньому.

Використовується зв’язок @OneToMany до Catalog, що відображає належність книг до певного каталогу.

@Setter(AccessLevel.PRIVATE)
@OneToMany(mappedBy = "catalog", cascade = CascadeType.ALL)
private List<Book> books = new ArrayList<>();

- Сутність Book зберігає дані про книги: назву, тіло книги, дату додавання книги в каталог, зв’язок із каталогом (Catalog) та нотатками до книги (рис. 3.2.)

```
@Entity 14 usages  Demchenko Vladislav *
@Table(name = "books")
@Setter
@Getter
@ToString(exclude = "notes")
@NoArgsConstructor
@EqualsAndHashCode
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book")
    @SequenceGenerator(name = "book", sequenceName = "book_seq", allocationSize = 1)
    private Long id;

    @Column(unique = true, nullable = false)
    private String name;

    private String body;

    @Temporal(TemporalType.DATE)
    private Date creationDate;

    @ManyToOne(optional = false)
    @JoinColumn(name = "catalog_fk")
    private Catalog catalog;
```

Рис. 3.2. Демонстрація Book сутності та її полів

Використовується зв’язок @ManyToOne до Catalog, що відображає належність книги до певної категорії.

@ManyToOne(optional = false)
@JoinColumn(name = "catalog_fk")
private Catalog catalog;

Використовується зв’язок @OneToMany до Note, що відображає належність нотаток до певної книги.

@JsonIgnore
@Setter(AccessLevel.PRIVATE)
@OneToMany(mappedBy = "book", cascade = CascadeType.ALL)
private List<Note> notes = new ArrayList<>();

- Сутність Note реалізує зберігання нотаток користувача. Пов'язана з книгою і залежить від неї, тобто не може існувати без книги окремо. Зберігає текст нотатки, дату створення та зв’язок з книгою

Використовується зв’язок @OneToMany до Note, що відображає належність нотаток до певної книги.

```
@ManyToOne(optional = false)
@JoinColumn(name = "book_id")
private Book book;
```

Реалізація моделей є максимально спрощеною, що відповідає принципу KISS (Keep It Simple) і забезпечує зручність подальшого використання моделей у сервісному та API-шарах.

У застосунку для взаємодії та доступу до сутностей бази даних використовується підхід, що ґрунтується на Spring Data JPA. Цей підхід дозволяє сфокусуватись на бізнес-логіці, делегуючи рутинні CRUD-операції фреймворку.

Репозиторії розміщено у пакеті repository, де для кожної сутності створено окремий інтерфейс, який розширює JpaRepository. Це надає доступ до повного набору стандартних методів, таких як findAll(), findById(), save(), deleteById() тощо та створення особистих ORM методів.

Приклад реалізації JpaRepository:

```
public interface BookRepository extends JpaRepository<Book, Long> {}
```

Окрім стандартних методів, реалізовані методи пошуку за параметрами моделей для використання в сервісах для бізнес-логіки.

```
List<Optional<Note>> findByBody(String body);
Optional<Catalog> findByCatalogName(String catalogName);
Optional<Book> findByName(String bookName);
```

Завдяки Spring Data JPA, уся логіка доступу до даних реалізована декларативно, без потреби створювати SQL-запити вручну. Це підвищує гнучкість і масштабованість застосунку, спрощуючи підтримку коду.

Якщо казати безпосередньо про взаємозв'язок між шаром доступу до даних (reposiory), який ми розглянули, та контролерами існує посередник - сервісний

рівень. Він відповідає за реалізацію бізнес-логіки, перевірку даних, обробку винятків і координацію взаємодії між об'єктами. Це дозволяє чітко розділити відповідальність між компонентами застосунку.

Щоб зменшити дублювання коду та забезпечити єдину контрактну структуру для всіх сервісів, було створено універсальний інтерфейс GlobalService<T>, який містить базові методи:

- save - відповідає за збереження T об’єкта

- delete - виконує видалення

- findAll - виконує пошук всіх елементів

Цей інтерфейс наслідується всіма конкретними сервісами - BookService, CatalogService, NoteService, змушуючи їх реалізовувати ці методи в своїх сервісах. Таким чином, кожен сервіс має єдину мінімально необхідну функціональність, яку гарантовано буде реалізовано. Також у нас з’являється можливість розширяти логіку за допомогою успадкування глобального інтерфейсу.

Крім того, кожен конкретний сервіс додає власні методи, специфічні до сутності, з якою він працює, що також відкриває можливості масштабування в рамках конкретних сутностей (рис. 3.2.).

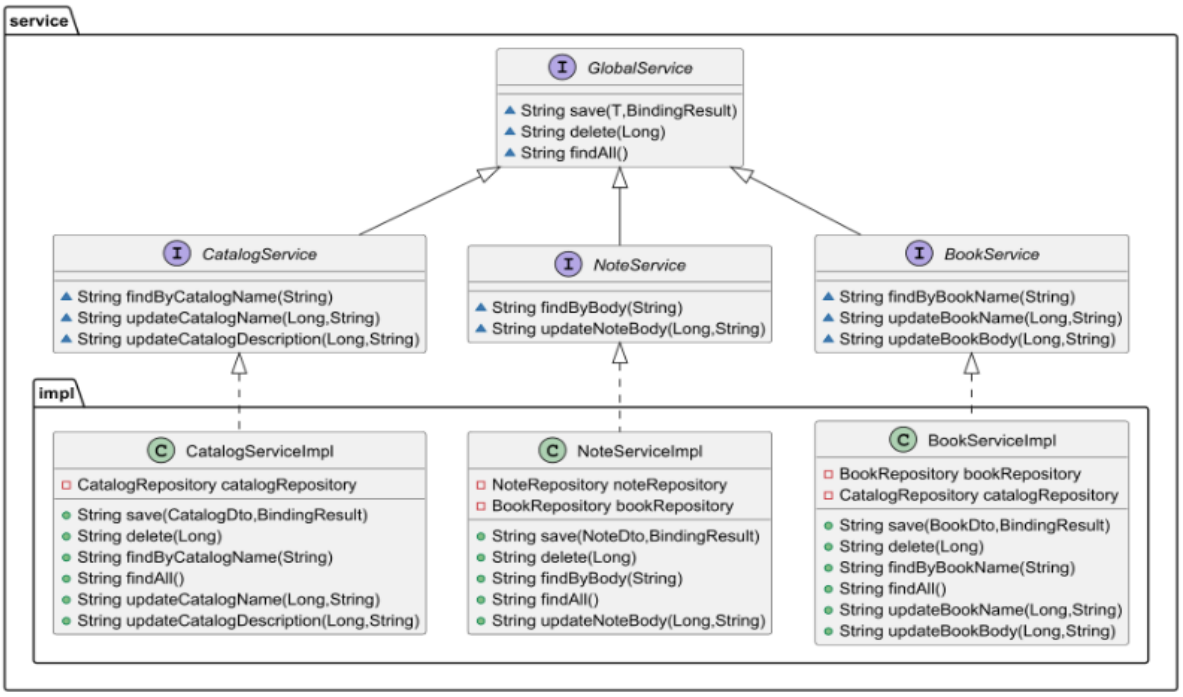


Рис. 3.2. Демонстраційна діаграма залежностей інтерфейсів та сервісів

Тобто на основі загального інтерфейсу GlobalService<T> було реалізовано три спеціалізовані сервіси, методи яких викликаються контролерами. Саме шар контролерів відповідають за отримання HTTP-запитів, обробку вхідних параметрів та передачу даних до відповідних сервісів. Цей підхід дозволяє чітко розділити відповідальність між представленням (API), бізнес-логікою (сервіси) та роботою з базою даних (репозиторії).

Усі контролери оформлено як REST-контролери за допомогою анотацій @RestController і обробників маршрутів, таких як @PostMapping, @GetMapping, @PutMapping, @DeleteMapping, що забезпечує реалізацію CRUD операцій в контролерах (рис. 3.3.).

```
@RestController  no usages  Demchenko Vladislav *
@RequestMapping(value = "/books")
@RequiredArgsConstructor
public class BookController {

    private final BookService bookService;
    @PostMapping("/create")  no usages  Demchenko Vladislav
    public ResponseEntity<?> addBook(@RequestBody @Valid BookDto bookDto, BindingResult bindingResult){
        return new ResponseEntity<>(bookService.save(bookDto, bindingResult), HttpStatus.CREATED);
    }

    @DeleteMapping("/delete/{id}")  no usages  Demchenko Vladislav
    public ResponseEntity<?> delete(@PathVariable Long id){
        return new ResponseEntity<>(bookService.delete(id), HttpStatus.OK);
    }


    @GetMapping("/findByName")  no usages  Demchenko Vladislav
    public ResponseEntity<?> findByName(@RequestParam String bookName){
        return new ResponseEntity<>(bookService.findByBookName(bookName), HttpStatus.OK);
    }

    @GetMapping("/findAll")  no usages  Demchenko Vladislav
    public ResponseEntity<?> getAllUsers(){
        return new ResponseEntity<>(bookService.findAll(), HttpStatus.OK);
    }

    @PutMapping("/updateName/{id}")  no usages  Demchenko Vladislav
    public ResponseEntity<?> updateName(@PathVariable Long id, @RequestParam String name){
        return new ResponseEntity<>(bookService.updateBookName(id, name), HttpStatus.OK);
    }

    @PutMapping("/updateBody/{id}")  no usages  Demchenko Vladislav
    public ResponseEntity<?> updateBody(@PathVariable Long id, @RequestParam String body){
        return new ResponseEntity<>(bookService.updateBookBody(id, body), HttpStatus.OK);
    }
}
```

Рис. 3.3. Приклад реалізації REST-контролера для Book об’єкта

Кожен контролер викликає відповідні методи свого сервісу, передаючи туди необхідні параметри. Це дозволяє відокремити логіку обробки запитів від бізнес-логіки, сприяючи чистій архітектурі та зручній підтримці коду.

Усі контролери дотримуються єдиного стилю побудови REST API, що забезпечує передбачуваність, узгодженість і зручність у використанні. Крім того, такий підхід дозволяє швидко масштабувати API шляхом додавання нових методів без порушення архітектури.

3.3. Інструменти, тестування та підтримка проєкту

З огляду на важливість стабільної роботи сервісного шару, наступним етапом стало його тестування. Для цього було використано бібліотеку Mockito, яка дає змогу створювати мок-об’єкти — штучні замінники справжніх залежностей. Це дозволило протестувати сервіси незалежно від реального підключення до бази даних, що спрощує перевірку бізнес-логіки (рис. 3.4.).

```
void findByBodyTest() {
    when(noteService.findByBody("body")).thenReturn( t: "found");
    ResponseEntity<?> response = noteController.findByBody("body");
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals( expected: "found", response.getBody());
}

@Test
void getAllNotesTest() {
    when(noteService.findAll()).thenReturn( t: "all");
    ResponseEntity<?> response = noteController.getAllNotes();
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals( expected: "all", response.getBody());
}
```

Рис. 3.4.. Приклад мок-тестів об’єкта Note

Під час тестування перевірялась коректність роботи ключових методів: збереження, оновлення, пошуку та видалення об'єктів. Крім того, тестувалась взаємодія між сервісами та репозиторіями, а також обробка помилок. Таким чином, тестування забезпечило впевненість у надійності реалізованої логіки, дозволивши вчасно виявити та усунути потенційні проблеми.

Усі тести були написані з урахуванням гнучкості, щоб їх було легко оновлювати при зміні логіки сервісів. Результати виконання тестів підтвердили правильність реалізації функціоналу та відповідність очікуваній поведінці.

ВИСНОВКИ

У межах кваліфікаційної роботи було розроблено вебзастосунок для створення та управління персональними каталогами книг. Основною метою стало впровадження інструменту, який дозволяє користувачеві впорядковувати власну бібліотеку, додавати до книг нотатки, об'єднувати їх у добірки, а також здійснювати контекстний пошук і фільтрацію вмісту. Реалізоване рішення має чітку багаторівневу архітектуру та відзначається високим ступенем гнучкості, розширюваності та готовності до практичного застосування.

Поставлені завдання для досягнення було виконано у повному обсязі.

Вивчено сучасні підходи до архітектури вебзастосунків, проаналізовано практику проєктування багаторівневих систем на Java із застосуванням фреймворку Spring Boot. Особливу увагу приділено принципам розділення відповідальностей між логічними шарами — контролер, сервіс, доступ до даних, що дозволило створити чітку й підтримувану структуру застосунку.

Визначено функціональні вимоги до системи, що дозволило сформувати перелік необхідного функціоналу: базові CRUD-операції для сутностей Catalog, Book та Note, можливість додавання коментарів, створення добірок, частковий пошук за назвою та описом. Також було враховано потребу у масштабованості системи та підтримці модульності, що відобразилося у виборі гнучкої архітектури та універсальних підходів до реалізації логіки.

Розроблено архітектурну модель застосунку, яка передбачає чіткий розподіл обов’язків між компонентами. Створено трьохрівневу структуру з окремими шарами: репозиторії для доступу до даних, сервіси для бізнес-логіки, контролери для обробки запитів. Завдяки цьому вдалося досягти високої зрозумілості коду, полегшити його підтримку та створити основу для масштабування.

Реалізовано функціонал фільтрації та контекстного пошуку, який дозволяє шукати книги за точним збігом в назві та описі. Такий підхід значно підвищує зручність користування системою, особливо у випадках великої кількості записів.

Здійснено тестування бізнес-логіки з використанням бібліотеки Mockito. Це дозволило перевірити ключові методи сервісного шару в ізоляції від бази даних, виявити й усунути потенційні помилки ще до впровадження, а також забезпечити стабільність функціонування всієї системи.

Оцінено наукову та практичну цінність створеного застосунку. Робота показала, що отримане рішення є не лише навчальним прикладом, а й практично придатним інструментом. Застосунок може бути використаний як приватним користувачем, так і адаптований для бібліотек, архівів, освітніх установ або невеликих організацій. Завдяки підтримці подальшого розширення — зокрема, додавання авторизації, інтеграції з зовнішніми сервісами, експорту та імпорту даних — система здатна еволюціонувати відповідно до потреб користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Баранов А. В. Практика тестування з Mockito. Системні дослідження та інформаційні технології. — 2021. — № 3. — С. 115–122.

2. Бачинський А. М., Кравець Р. Б., Пилипчук О. П. Програмування на Java. Львів: Видавництво "Бескид Біт", 2015.

3. Бойко П. С. Java для професіоналів: сучасні рішення. — Харків: Фактор, 2021. — 336 с.

4. Бойчук Л.В., Музика О.М. Інтернет в маркетингу: Підручник. — К.: Центр учбової літератури, 2010. — 512 с.

5. Бондаренко О. В. Архітектура програмних систем: навч. посібник. — Харків: ХНУРЕ, 2018. — 192 с.

6. Бублик В. В. Об'єктно-орієнтоване програмування. Харків: Видавництво "Веста", 2016.

7. Бублик В. В., Костюк О. М., Тулашвілі Ю. Й. Інформаційні технології в професійній діяльності. Київ: Видавничий центр "Академія", 2012.

8. Вовк О.Б. Методи та засоби підвищення життєздатності вебсайту як інформаційного продукту: автореф. дис. ... канд. техн. наук: 05.13.06 / Олена Борисівна Вовк. — Л., 2013. — 20 с.

9. Гамма Е. та ін. Шаблони проєктування. — К.: Діалектика, 2020. — 367 с.

10. Гардабацька І.А. Тенденції розвитку дизайну доби постіндустріального суспільства. Вісник КНУКіМ. Серія: Мистецтвознавство. — 2019. — № 41. — С. 176–184.

11. Глибовець М. М., Олецький О. В., Пасічник В. В. Основи баз даних та інформаційних систем. Львів: Видавничий центр ЛНУ імені Івана Франка, 2003.

12. Гриненко Н. Архітектура програмного забезпечення: навчальний посібник. — К.: КНУ, 2020. — 168 с.

13. Грицак О.В. Веб-технології та дизайн. Р.В. Мощич. Вісник Хмельницького національного університету. Технічні науки. — 2021. — № 1. — С. 22–26.

14. Завацький О. Ю. Бази даних та інформаційні системи. Київ: Видавничий дім "Професіонал", 2007.

15. Кравець М.В. Патерни проєктування у Java: методичні рекомендації. — Київ: НТУУ «КПІ», 2020. — 72 с.

16. Кудінов Д. Ю. Інформаційні системи та технології: підручник. — К.: Ліра-К, 2019. — 328 с.

17. Кузьменко О. М. Практика сучасної Java-розробки. — Київ: Видавництво «Ліра-К», 2021. — 236 с.

18. Литвин О. С., Павлюк І. М. Архітектура веборієнтованих систем // Збірник наукових праць НУ «Львівська політехніка». — 2022. — № 75. — С. 55–63.

19. Паньков С. І. Основи юніт-тестування в Java. — Київ: Університетська книга, 2020. — 216 с.

20. Романчиков В. І. Основи Web-програмування. Київ: Видавництво "Центр учбової літератури", 2011.

21. Семенов С. В., Трофименко О. Г. Алгоритми та структури даних. Київ: Видавництво "Либідь", 2012.

22. Соловей О.Ю. Архітектура веб-додатків. — Львів: ЛНУ імені І. Франка, 2021. — 156 с.

23. Сорокіна Н. М. Програмування веб-застосунків: навч. посіб. — Львів: Новий Світ–2000, 2020. — 224 с.

24. Трофименко О. Г., Яворський В. В., Левченко Т. В. Інформатика та комп'ютерна техніка. Київ: Видавництво "Знання", 2010.

25. Фаулер М. Архітектура корпоративних програмних застосунків. — К.: Діалектика, 2021. — 528 с.

26. Шевченко І. В. Сучасні системи керування збіркою в Java-проєктах // Вісник ХНУРЕ. — 2022. — № 4. — С. 50–57.

27. Шеремет А. Ю. Застосування Lombok у Spring Boot-проєктах // Збірник наукових праць НТУУ "КПІ". — 2022. — № 5. — С. 85–90.

28. Шостак О. Основи Spring Framework. — Харків: ФОП Панов, 2022. — С. 112–140.

29. Шумейко Л. І. Автоматизація програмного забезпечення та DevOps в Java-проєктах. Науковий вісник КНУ. — 2022. — № 11. — С. 57–122.

30. Bass, L., Clements, P., & Kazman, R. Software architecture in practice. Addison-Wesley Professional, 2012.

31. Coulouris, G., Dollimore, J., & Kindberg, T. Distributed systems: concepts and design. Addison-Wesley, 2011.

32. Daigneau R. Designing Web APIs with Strive for Simplicity. — Addison-Wesley, 2022. — 304 p.

33. Fowler, M. Patterns of enterprise application architecture. Addison-Wesley Professional, 2012.

34. Hohpe, G., & Woolf, B. Enterprise integration patterns: designing, building, and deploying messaging solutions. Addison-Wesley Professional, 2003.

35. Newman, S. Building microservices: designing fine-grained systems. O'Reilly Media, 2015.

36. Richards M. Microservices vs. Service-Oriented Architecture. — O’Reilly Media, 2016. — 54 p.

37. Sommerville, I. Software engineering. Pearson, 2016.

38. Tanenbaum, A. S., & Van Steen, M. Distributed systems: principles and paradigms. Pearson Prentice Hall, 2007.

39. The Importance of Separation of Concerns in Software Engineering. Medium. Retrieved from. URL: https://medium.com/@communication_93652/the-importance-of-separation-of-concerns-in-software-engineering-8d5964ba65d9.