

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Факультет технологій та інформаційних систем

Кафедра інформаційних технологій та систем

Майнаєв Фарідун Якубджонович

**РЕАЛІЗАЦІЯ ЗАХИСТУ ВІД SQL-ІН'ЄКЦІЙ У ВЕБДОДАТКУ
ЗАСОБАМИ PYTHON, DJANGO**

кваліфікаційна робота

здобувача вищої освіти першого (бакалаврського) рівня
освітньої програми «Інженерія програмного забезпечення»
за спеціальністю 121 Інженерія програмного забезпечення

Особистий підпис



Фарідун МАЙНАЄВ

Науковий керівник



Світлана ПЕРЕЯСЛАВСЬКА,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри



Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Міністерство освіти і науки України

Державний заклад «Луганський національний університет
імені Тараса Шевченка»

Факультет (інститут)

Навчально-науковий інститут математики та
інформаційних технологій

(повна назва)

Кафедра

Інформаційних технологій та систем

(повна назва)

Освітній ступень

Бакалавр

(код, назва)

Напрямок підготовки

Комп'ютерна інженерія

(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТС

М. А. Семенов

(підпис)

(ініціали, прізвище)

“ ”

2025 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТУ

Майнаєву Фарідуну Якубджоновичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи)

Реалізація захисту від SQL-ін'єкцій у
вебдодатку засобами Python, Django.

Керівник кваліфікаційної роботи

Переяславська С.О. к.п.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом по університету

від

2. Строк подання студентом проекту (роботи)

3. Вихідні дані до роботи (проекту)

Реалізація захисту від

SQL-ін'єкцій у вебдодатку засобами Python, Django. вебдодатку засобами
Python, Django.

(визначаються кількісні або (та) якісні показники, яким повинен відповідати об'єкт розробки)

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно
розробити)

РОЗДІЛ 1. Аналіз проблеми SQL-ін'єкцій у веб-додатках

РОЗДІЛ 2. Методи та засоби захисту від SQL-ін'єкцій

РОЗДІЛ 3. Проектування та створення вебдодатку

РОЗДІЛ 4. Тестування та аналіз безпеки вебдодатку

(визначаються назви розділів або (та) перелік питань, які повинні увійти до тексту ПЗ)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Структурна схема веб-застосунку; діаграма варіантів використання; схема
рівнів захисту; порівняльна таблиця результатів тестування.

6. Консультанти розділів проекту/роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання „_____” _____ 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
	Вибір теми роботи, вивчення наукової літератури, затвердження теми та керівника.	До 15 жовтня	
	Аналіз літературних джерел за темою роботи. Розробка та апробація методики дослідно-експериментальної роботи. Подання структури теоретичної частини роботи та плану експериментальних досліджень.	Другий тиждень листопада (10 листопада)	
	Робота над теоретичною частиною. Подання теоретичної частини роботи для першого читання науковим керівником.	До 15 грудня	
	Усунення зауважень, урахування рекомендацій наукового керівника. Подання теоретичної частини роботи на друге читання.	До 28 січня	
	Проведення експериментальної роботи. Поетапний аналіз та обговорення її результатів. Перевірка стану виконання роботи.	Перший тиждень березня	
	Урахування рекомендацій наукового керівника, усунення недоліків, підготовка варіанта роботи до передзахисту. Розробка презентації.	До 31 березня	
	Попередній захист роботи на кафедрі	квітень	
	Доопрацювання роботи з урахуванням рекомендацій після передзахисту. Подання роботи науковому керівникові та рецензентові на підготовку відгуку та рецензії	За 10 днів до державної атестації	
	Подання на кафедру остаточного варіанта роботи, переплетеного та підписаного автором, науковим керівником і рецензентом.	За 5 днів до державної атестації	

Студент

Керівник проекту (роботи)

[Підпис]

підпис

Ф. Я. Майнаєв

(ініціали, прізвище)

С. О. Переяславська

(ініціали, прізвище)

АНОТАЦІЯ

Майнаєв Фарідун Яубджонович

Тема: Реалізація захисту від SQL-ін'єкцій у вебдодатку засобами Python, Django.

Спеціальність: 121 «Інженерія програмного забезпечення»

Установа: ДЗ «Луганський національний університет імені Тараса Шевченка», 2026 р.

Бакалаврська робота містить: 97 с., 10 рис., 13 табл., 21 лістинг, 22 джерела, 6 додатків.

Об'єкт дослідження – процес розробки вебдодатку та його програмних компонентів, які забезпечують його взаємодію з реляційною базою даних.

Предмет дослідження – методи та засоби захисту вебдодатків від SQL-ін'єкцій у середовищі Python та Django.

Мета роботи – реалізація механізмів захисту вебдодатку від SQL-ін'єкцій із використанням мови програмування Python та фреймворку Django.

Результати роботи. У роботі реалізовано вебзастосунок управління розкладом для навчальних закладів з десятьма незалежними рівнями захисту від SQL-ін'єкцій: ORM-параметризація запитів через Django, власний SQLInjectionValidator (11 регулярних виразів), захист параметрів сортування (sort_map whitelist), CSRF-захист, рольова авторизація (адміністратор / викладач / студент), валідатор складності пароля (PasswordComplexityValidator), безпека сесій (HttpOnly, SameSite cookies), захист від Clickjacking (X-Frame-Options: DENY), мінімізація привілеїв бази даних та журналювання подій безпеки (security.log). Тестування охопило 19 маршрутів і 14 SQL-векторів атак: SQLInjectionValidator виявив 100 % шкідливих payload-ів при 0 % хибних спрацьовувань, усі POST-запити без CSRF-токена отримали HTTP 403.

Ключові слова. SQL-ін'єкція, Django, Python, ORM, SQLInjectionValidator, захист вебзастосунків, кібербезпека, валідація даних, CSRF, рольова авторизація, Defense-in-Depth.

ABSTRACT

Mainaiev Faridun Yaubdzhonovich

Topic: Implementation of SQL Injection Protection in a Web Application Using Python, Django.

Speciality: 121 "Software Engineering"

Institution: Luhansk Taras Shevchenko National University, 2026.

The diploma work contains: 97 pages, 10 figures, 13 tables, 21 listings, 22 references, 6 appendices.

Object of research – the process of developing a web application and its software components that ensure interaction with a relational database.

Subject of research – methods and tools for protecting web applications from SQL injections in the Python and Django environment.

Purpose of the work – implementation of web application protection mechanisms against SQL injections using Python programming language and Django framework.

Results. A schedule management web application for educational institutions was developed with ten independent SQL injection protection layers: Django ORM query parameterization, custom SQLInjectionValidator (11 regular expressions), sort parameter whitelist (sort_map), CSRF protection, role-based authorization (admin / teacher / student), password complexity validator (PasswordComplexityValidator), session security (HttpOnly, SameSite cookies), Clickjacking protection (X-Frame-Options: DENY), database privilege minimization, and security event logging (security.log). Testing covered 19 routes and 14 SQL attack vectors: SQLInjectionValidator detected 100% of malicious payloads with 0% false positives; all POST requests without a CSRF token received HTTP 403.

Keywords. SQL injection, Django, Python, ORM, SQLInjectionValidator, web application security, cybersecurity, data validation, CSRF, role-based authorization, Defense-in-Depth.

Міністерство освіти і науки України
Державний заклад «Луганський національний університет
імені Тараса Шевченка»

Факультет (інститут)

Факультет технологій та інформаційних
систем

(повна назва)

Кафедра

Інформаційних технологій та систем

(повна назва)

(код, назва)

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання програмної розробки (ПР):

РЕАЛІЗАЦІЯ ЗАХИСТУ ВІД SQL-ІН'ЄКЦІЙ У ВЕБДОДАТКУ
ЗАСОБАМИ PYTHON, DJANGO
ІТС.ІПЗ-4.1826-02-ТЗ

ПОГОДЖЕНО

Керівник кваліфікаційної роботи

Переяславська С.О.



“ _____ ” 2026р

ВИКОНАВЕЦЬ

Студент групи ІПЗ-4

Майнаєв Ф.Я.



“ _____ ” 2026р

ЗМІСТ

ВСТУП.....	3
1. ХАРАКТЕРИСТИКА ОБ'ЄКТА.....	3
2. ПРИЗНАЧЕННЯ ТОВАРІВ.....	3
3. ОСНОВНІ ВИМОГИ ДО ПРОГРАМНОГО КОМПЛЕКСУ.....	4
4. ТЕХНІКО-ЕКОНОМІЧНІ ВИМОГИ ДО КІНЦЕВОГО ПРОДУКТУ.....	5
5. ВИМОГИ ДО МАТЕРІАЛІВ І КОМПЛЕКТУЮЧИХ.....	5
6. ЕТАПИ ВИКОНАННЯ ПР.....	5
7. ПРИЙОМ.....	6
8. ПОРЯДОК ВНЕСЕННЯ ЗМІН ДО ТЕХНІЧНОГО ЗАВДАННЯ, ЩО ЗАТВЕРДЖЕНО.....	7

ВСТУП

1.1. Найменування: Вебдодаток управління розкладом для навчальних закладів із захистом від SQL-ін'єкцій.

1.2. Шифр ПР: ІТС.ІПЗ-4.1826-02-ТЗ

1.3. Підстава до виконання ПР: Підставою для виконання даної розробки є необхідність реалізації безпечного вебдодатку управління розкладом занять навчального закладу з комплексним захистом від SQL-ін'єкцій засобами Python та Django.

1.4. Терміни розробки:

1.4.1. Початок 1 жовтня 2025 р.

1.4.2. Закінчення 30 травня 2026 р.

1.5. Фінансується за рахунок коштів замовника.

1. ХАРАКТЕРИСТИКА ОБ'ЄКТА

1.1. Об'єктом розробки є вебдодаток управління розкладом навчального закладу. До складу об'єкту повинно входити:

1.1.1. Серверна частина (бекенд) на Python 3.13 / Django 5 з базою даних PostgreSQL.

1.1.2. Клієнтська частина (фронтенд) на основі Bootstrap 5.3 та шаблонізатора Django Templates.

1.1.3. Система безпеки з комплексом механізмів захисту від SQL-ін'єкцій та журналюванням подій безпеки.

1.2. **До вхідної інформації** належать дані про розклад занять, облікові записи студентів, викладачів та адміністраторів навчального закладу.

2. ПРИЗНАЧЕННЯ ТОВАРІВ

2.1. Призначення: централізоване управління розкладом занять навчального закладу із забезпеченням цілісності та конфіденційності даних.

2.2. Основні критерії ефективності:

2.2.1. Студент повинен мати можливість переглядати актуальний розклад занять з фільтрацією за датою, групою та викладачем.

2.2.2. Викладач повинен мати можливість переглядати розклад із фільтрацією за датою, групою та аудиторією.

2.2.3. Адміністратор повинен мати можливість повноцінного управління розкладом, користувачами та перегляду журналу подій безпеки.

2.2.4. Система повинна автоматично виявляти та блокувати спроби SQL-ін'єкцій, фіксуючи їх у журналі безпеки.

3. ОСНОВНІ ВИМОГИ ДО ПРОГРАМНОГО КОМПЛЕКСУ

3.1. Загальні вимоги

3.1.1. Вебдодаток працює на операційній системі Windows та Linux.

3.1.2. Система підтримує одночасну роботу кількох користувачів; час відповіді сервера не перевищує двох секунд при стандартному навантаженні.

3.1.3. Вебдодаток повинен мати адаптивний інтерфейс, що коректно відображається на різних розмірах екранів.

3.2. Додаткові вимоги

3.2.1. Мова програмування Python версії не нижче 3.10; фреймворк Django версії 5 або вище.

3.2.2. СУБД PostgreSQL; допускається використання SQLite під час розробки.

3.2.3. Вимоги до ліцензійного програмного забезпечення не передбачаються.

3.3. Вимоги до складу і архітектури

3.3.1. Використання клієнт-серверної архітектури та модульної структури додатку (MVT).

3.3.2. Ізоляція бізнес-логіки від представлення; взаємодія з БД виключно через ORM.

3.3.3. Заборона використання сирих SQL-запитів без параметризації.

3.4. Вимоги до якості і надійності

3.4.1. Захист від SQL-ін'єкцій: SQLInjectionValidator (11 регулярних виразів), whitelist-словник sort_map для ORDER BY, CsrfViewMiddleware, рольова авторизація, PasswordComplexityValidator.

3.4.2. Логування подій безпеки у базі даних із записом типу події, IP-адреси та часової мітки (SecurityLog).

3.4.3. Коректна обробка помилок введення даних; збереження цілісності даних у разі збоїв.

3.5. Вимоги до експлуатації

3.5.1. Апаратне забезпечення: процесор не менше двох ядер, оперативна пам'ять не менше 4 ГБ.

3.5.2. Розробник гарантує роботу вебдодатку без збоїв та переналаштувань після встановлення.

4. ТЕХНІКО-ЕКОНОМІЧНІ ВИМОГИ ДО КІНЦЕВОГО ПРОДУКТУ

Вартість робіт по розробці даної ПР визначається згідно з договором на розробку. Вартість пропонувананих аналогів повинна забезпечити економічну доцільність їх застосування.

5. ВИМОГИ ДО МАТЕРІАЛІВ І КОМПЛЕКТУЮЧИХ

5.1. Вимоги до екологічної безпеки при експлуатації. Не пред'являються.

5.2. Спеціальні вимоги до кінцевого продукту. Не пред'являються.

5.3. Вимоги до безпеки для населення при експлуатації продукції. Не пред'являються.

6. ЕТАПИ ВИКОНАННЯ ПР

Етапи виконання ПР можуть уточнюватися згідно календарного плану робіт за погодженням між замовником і виконавцем.

№	Назва етапу виконання роботи	Термін виконання	Звітні матеріали
1	Аналіз методів SQL-ін'єкцій та засобів захисту. Проектування інфологічної та даталогічної моделей бази даних.	Жовтень – грудень 2025 р.	Фрагмент програмного комплексу, що реалізує базову структуру проєкту; звітна документація.
2	Реалізація SQLInjectionValidator, захисту ORDER BY (sort_map) та системи журналювання SecurityLog. Розробка функціоналу управління розкладом та інтерфейсу.	Січень – березень 2026 р.	Готовий програмний комплекс на ЕОМ замовника.

3	Тестування на векторах атак OWASP Top 10. Аналіз результатів. Оформлення пояснювальної записки та звітних матеріалів.	Квітень – травень 2026 р.	Звітні матеріали згідно з пунктом 8 цього ТЗ.
---	---	---------------------------	---

7. ПРИЙОМ

7.1. Необхідні вимоги для впровадження ПР і завершення робіт.

Оцінка результатів розробки та доцільність її продовження здійснюється замовником за поданням наступних матеріалів:

- встановлений та функціональний веб-додаток управління розкладом;
- перелік файлів на резервному носії;
- короткий опис роботи ПР та опис усіх файлів, необхідних для роботи;
- перелік документів: Технічне завдання; Пояснювальна записка; Програма і методика тестування.

7.2. Перелік звітних документів, необхідних для прийняття етапів роботи:

- короткий опис результатів кожного етапу у вигляді анотованого звіту;
- частковий програмний комплекс на ЕОМ замовника згідно календарного плану робіт;
- акт приймання продукції.

Звітні матеріали подаються у вигляді звітів на папері.

7.3. Загальний перелік до прийому звітних документів.

До прийому пред'являються: акт здачі-приймання продукції, акт впровадження ПР.

7.4. Тестування ПР.

Тестування виконується згідно з «Програмою і методикою тестування», яка розробляється виконавцем та затверджується замовником. Тестовий план включає:

- функціональне тестування 19 маршрутів застосунку;

- тестування SQLInjectionValidator: перевірка 14 SQL-векторів атак на 17 полях у 8 групах форм – очікується 100 % виявлення шкідливих payload-ів та 0 % хибних спрацьовувань;
- тестування PasswordComplexityValidator: не менше 9 тест-кейсів з перевіркою граничних значень;
- тестування CSRF-захисту: усі POST-запити без токена мають отримувати HTTP 403;
- тестування рольової авторизації: перевірка неможливості горизонтального підвищення привілеїв;
- аналіз результатів тестування та порівняння рівня захищеності до та після впровадження механізмів безпеки.

8. ПОРЯДОК ВНЕСЕННЯ ЗМІН ДО ТЕХНІЧНОГО ЗАВДАННЯ, ЩО ЗАТВЕРДЖЕНО

Дане технічне завдання може уточнюватися в процесі розробки ПР при узгодженні сторін з оформленням доповнень до ТЗ.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЗ «ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Факультет технологій та інформаційних систем

(назва факультету, інституту)

Інформаційних технологій та систем

(назва кафедри)

Пояснювальна записка
до дипломного проєкту (роботи)
БАКАЛАВРА
(освітньо-кваліфікаційний рівень)

на тему:
РЕАЛІЗАЦІЯ ЗАХИСТУ ВІД SQL-ІН'ЄКЦІЙ У ВЕБДОДАТКУ
ЗАСОБАМИ PYTHON, DJANGO

Виконав: студент 4 курсу, групи ІІЗ-4
напряму підготовки (спеціальності)
121 «Інженерія програмного забезпечення»
(шифр і назва напряму підготовки, спеціальності)

Майнаєв Ф.Я.
(прізвище та ініціали)

Керівник Переяславська С.О.
(прізвище та ініціали)

Рецензент Козуб Ю. Г.
(прізвище та ініціали)

Лубни – 2026 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ		4-6
ВСТУП		7
РОЗДІЛ 1	Аналіз проблеми SQL-ін'єкцій у вебдодатках	8-20
	1.1. Аналіз проблеми безпеки вебзастосунків	8
	1.2. Поняття, принципи роботи та класифікація SQL-ін'єкцій	10
	1.3. Аналіз сучасних інцидентів та статистики вразливостей	16
	Висновки до розділу 1	19
РОЗДІЛ 2	Методи та засоби захисту від SQL-ін'єкцій	21-35
	2.1. Загальні принципи безпечної розробки вебдодатків	21
	2.2. Підготовлені запити та ORM	23
	2.3. Валідація та санітизація вхідних даних	27
	2.4. Механізми захисту у фреймворці Django	31
	Висновки до розділу 2	34
РОЗДІЛ 3	Проектування та створення вебдодатку	36-78
	3.1. Вибір технологічного стеку та архітектури вебдодатку	36
	3.2. Проектування бази даних	49
	3.3. Реалізація функціоналу вебдодатку	56
	3.4. Реалізація механізмів захисту від SQL-ін'єкцій	64
	Висновки до розділу 3	77
РОЗДІЛ 4	Тестування та аналіз безпеки вебдодатку	78-92
	4.1. Методика тестування вебдодатку на SQL-ін'єкції .	78
	4.2. Проведення тестів та аналіз результатів	82

	4.3. Порівняння рівня безпеки до та після впровадження захисту	84
	4.4. Оцінка ефективності реалізованих методів	87
	Висновки до розділу 4	90
ЗАГАЛЬНІ ВИСНОВКИ		93-94
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ		95-97
ДОДАТКИ		98

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД	база даних
СУБД	система управління базою даних
ШІ	штучний інтелект
AJAX	Asynchronous JavaScript and XML — асинхронні запити на сервер без перезавантаження сторінки
API	Application Programming Interface – програмний інтерфейс застосунку
CI/CD	Continuous Integration / Continuous Deployment – безперервна інтеграція та безперервне розгортання
CISA	Cybersecurity and Infrastructure Security Agency – Агентство кібербезпеки та безпеки інфраструктури (США)
CSP	Content Security Policy – політика безпеки контенту
CSRF	Cross-Site Request Forgery – підробка міжсайтових запитів
CRUD	Create, Read, Update, Delete – базові операції зі збереженими даними (створення, читання, оновлення, видалення)
CVE	Common Vulnerabilities and Exposures – база даних загальних вразливостей та впливів
CVSS	Common Vulnerability Scoring System – загальна система оцінки вразливостей
DAST	Dynamic Application Security Testing – динамічне тестування безпеки застосунків
DBIR	Data Breach Investigations Report
DDL	Data Definition Language — мова визначення даних (підмножина SQL для структури бази даних)
DDoS	Distributed Denial of Service – розподілена атака на відмову в обслуговуванні

DML	Data Manipulation Language – мова маніпулювання даними (підмножина SQL)
DTL	Django Template Language – мова шаблонів фреймворку Django
FK	Foreign Key – зовнішній ключ у реляційній базі даних
HTML	HyperText Markup Language – мова гіпертекстової розмітки
HTTP	HyperText Transfer Protocol – протокол передачі гіпертексту
HTTPS	HyperText Transfer Protocol Secure – захищений протокол передачі гіпертексту
HSTS	HTTP Strict Transport Security – механізм захисту, що змушує браузер використовувати лише HTTPS
IDOR	Insecure Direct Object Reference – небезпечне пряме посилання на об'єкт (вразливість авторизації)
LLM	Large Language Model – велика мовна модель (тип моделі штучного інтелекту)
MVT	Model-View-Template – архітектурний шаблон фреймворку Django
ORM	Object-Relational Mapping – об'єктно-реляційне відображення; технологія перетворення між об'єктами застосунку та таблицями бази даних
OWASP	Open Web Application Security Project – відкритий проєкт безпеки веб-застосунків
SAST	Static Application Security Testing – статичне тестування безпеки застосунків (аналіз вихідного коду)
SQL	Structured Query Language – мова структурованих запитів для взаємодії з реляційними базами даних
UML	Unified Modeling Language – уніфікована мова моделювання
URL	Uniform Resource Locator – уніфікований покажчик ресурсу (адреса в мережі)

XSS	Cross-Site Scripting – міжсайтовий скриптинг (атака через упровадження шкідливих скриптів)
ZAP	Zed Attack Proxy – інструмент OWASP для автоматизованого тестування безпеки вебзастосунків

ВСТУП

Безпека додатків загалом та вебдодатків зокрема є однією з ключових проблем у сучасній розробці програмного забезпечення. Наразі в значній частині вебдодатків для зберігання й обробки інформації використовуються реляційні бази даних. Хоча це й має свої переваги, але робить систему вразливою до таких типів атак як SQL-ін'єкції. SQL-ін'єкції надають зловмисникам можливість обійти автентифікацію й отримати несанкціонований доступ до даних, що може призвести, окрім витоку даних, до їх модифікації або видалення.

Незважаючи на наявність фреймворків і засобів безпеки, проблематика SQL-ін'єкції все ще залишається актуальною через можливі помилки під час проєктування та реалізації вебдодатків. Навіть незначні помилки принципів безпечної розробки можуть поставити під удар цілісність та конфіденційність даних користувачів, що в свою чергу несе фінансові та репутаційні ризики для компаній. Саме тому дослідження й упровадження ефективних механізмів захисту від SQL-ін'єкцій є важливим завданням у сфері розробки додатків.

Метою даної бакалаврської роботи є реалізація механізмів захисту вебдодатку від SQL-ін'єкцій із використанням мови програмування Python та фреймворку Django. Для її досягнення в межах роботи розроблено вебдодаток, що демонструє безпечні підходи до взаємодії з базою даних й обробки вхідних даних. Об'єкт роботи – процес розробки вебдодатку та його програмних компонентів, які забезпечують його взаємодію з реляційною базою даних. Предмет дослідження – методи та засоби захисту вебдодатків від SQL-ін'єкцій у середовищі Python та Django. Методи дослідження включають аналіз наукових і технічних джерел із питань безпеки вебдодатків, дослідження типових сценаріїв SQL-ін'єкцій, використання вбудованих механізмів захисту фреймворку Django, проєктування, розроблення та тестування вебдодатку. Практична цінність роботи полягає в створенні вебдодатку з реалізованими механізмами захисту від SQL-ін'єкцій, який може бути використаним як у якості навчального зразка, так і в якості основи для створення безпечних вебсистем.

РОЗДІЛ 1

АНАЛІЗ ПРОБЛЕМ SQL-ін'єкцій у вебдодатках

1.1. Аналіз проблеми безпеки вебзастосунків

У сучасному цифровому середовищі роль вебзастосунків важко переоцінити: вони використовуються для забезпечення надання інформації, послуг, взаємодії між окремими людьми та групами людей, є ефективними інструментами при веденні бізнесу. Зростання їх популярності не могло не призвести до збільшення рівня кіберзагроз і кіберзлочинності, спрямованих проти цих систем. Безпека вебзастосунків одночасно є проблемою розробників, адміністраторів та звичайних користувачів [1, с. 18].

Аналіз звітів провідних організацій у сфері кібербезпеки, таких як Open Web Application Security Project (OWASP), Veracode, Check Point свідчать про щорічне зростання кількості атак на вебзастосунки. Причин декілька: недбалий процес розробки, відсутність тестування безпеки, застарілі фреймворки, некоректно налаштовані сервери тощо [1, с. 18]. Згідно з даними рейтингу OWASP Top 10, SQL-ін'єкції стабільно входять до переліку найкритичніших загроз для вебзастосунків [2, с. 1].

Найпоширенішими загрозами є такі: мова структурованих запитів для взаємодії з реляційними базами даних (SQL-ін'єкції), міжсайтовий скриптинг (XSS), підробка міжсайтових запитів (CSRF), атака у відмові в обслуговуванні (DDoS), несанкціонований доступ через незахищений програмний інтерфейс застосунку (API) та захоплення сеансів. Із цього списку окремо слід виділити SQL-ін'єкції: за їх допомогою зловмисник має змогу ввести шкідливий код у запит до бази даних. Успішна SQL-ін'єкція призводить до компрометації бази даних: витоку конфіденційних даних, їх зміни або видалення [3, с. 1].

Для вебсистем SQL-ін'єкція є однією з найпоширеніших та найруйнівніших загроз. Для виявлення цієї вразливості використовують декілька методів: аналіз (статичний, гібридний, динамічний), методи машинного навчання та інтелектуального аналізу даних [2, с. 2]. Сучасні фреймворки містять

вбудовані механізми захисту – апараметризовані запити, об’єктно-реляційне відображення (ORM) та валідацію введених даних. Проте для проєктів, реалізованих на застарілому стеці або з великою кількістю власного коду, проблема залишається актуальною [1, с. 19].

Важливим чинником вразливості є людський фактор. Розробник може припуститися помилки через незнання базових принципів захисту або нехтування ними: використовувати динамічні SQL-запити без параметризації, не обмежувати доступ. Низький рівень обізнаності розробників щодо потенційних загроз і методів захисту від них є однією з основних причин успішності цих атак [3, с. 4]. До того ж, тиск установлених дедлайнів нерідко призводить до відсутності належної уваги до етапу тестування, що призводить, у свою чергу, до виходу в продакшн продукту зі значними недоліками в безпеці [2, с. 4].

Для виявлення вразливостей у вебзастосунках використовуються автоматизовані сканери, наприклад, Acunetix, Burp Suite, Nessus, SQLMap тощо [4, с. 4]. Але слід зазначити, що деякі науковці, наприклад, Замрій і Шахматов зазначають, що хоч сучасні автоматизовані сканери безпеки (OWASP, Zed Attack Proxy (ZAP), SQLMap) і демонструють прийнятні результати при виявленні SQL-ін’єкцій і XSS, проте часто ігнорують інші вразливості зі списку OWASP Top 10, що є підтвердженням необхідності комплексного підходу до тестування безпеки вебдодатків [5, с. 59].

Сам процес тестування починається з аналізу архітектури додатку та його функціональності. Автоматизовані інструменти та фахівці імітують дії потенційного зловмисника, використовуючи різні методи для обходу автентифікації. Одним із підходів є динамічне тестування (DAST). За цим підходом, до запущеного застосунку відправляються шкідливі запити й аналізується реакція сервера [1, с. 22]. Щоб дослідити код, до його виконання застосовується статичне тестування безпеки застосунків (SAST). Цей аналіз дає змогу виявити вразливість ще на стадії ранньої розробки [1, с. 22].

Ефективне тестування передбачає як автоматизовані інструменти, так і ручну перевірку. За своєю суттю воно є багатоетапним. Автоматизовані запити

необхідні для охоплення великої кількості SQL-адрес та форм, а ручне тестування необхідне для складних логічних тестів та аналізу функціональності [4, с. 6]. Вразливою також є перевірка автентифікації та управління сесіями – у разі відсутності або ж обробки токенів сесії неналежним чином, зломисник отримує можливість отримати контроль над обліковими записами користувачів [3, с. 3].

Поруч із класичними методами тестування використовуються підходи на основі машинного навчання для виявлення вразливостей у коді. Наприклад, дослідження рекурентних нейронних архітектур (peerhole LSTM, Gated Recurrent Unit тощо) у задачах виявлення SQL-ін'єкцій у коді Python показали, що модель peerhole LSTM досягає показника $F1=0,90$, що вище за попередні дослідження приблизно на 10% [6, с. 14].

Для реалізації комплексного захисту вебзастосунків використовується принцип «глибинного ешелюваного захисту» (Defense in Depth), який передбачає багаторівневу систему безпеки: валідацію даних (як на стороні клієнта, так і на стороні сервера), шифрування, обмеження доступу та логування. Підхід DevSecOps здатен забезпечити раннє виявлення загроз за рахунок інтеграції безпеки безпосередньо в життєвий цикл розробки. А такі інструменти як GitHub Security, GitLab Secure та Snyk дозволяють розробникам отримувати попередження про небезпечні бібліотеки ще в середовищі розробки [1, с. 22].

Отже, сучасний стан безпеки вебзастосунків характеризується з одного боку розвитком загроз, а з іншого – пошуком нових методів захисту. Безпека вебзастосунків є багатоетапним процесом. Комплексний підхід, який поєднує автоматизоване та ручне тестування, DevSecOps-практики та навчання персоналу, надає можливість знизити ризики зламу та забезпечити надійний захист від актуальних загроз [4, с. 7].

1.2. Поняття, принципи роботи та класифікація SQL-ін'єкцій

SQL – це стандартна декларативна мова програмування, яка необхідна для взаємодії з базами даних. Використовується SQL для операцій вставки, вибірки, оновлення та видалення інформації з бази даних (БД). Нині переважна більшість

вебдодатків використовують реляційні СУБД для роботи з даними, через що SQL став невід’ємною складовою архітектури багатьох вебдодатків, що у свою чергу робить сучасні вебдодатки вразливими для SQL-ін’єкцій.

SQL-ін’єкція – тип атаки, яка дає зловмиснику можливість вставити або додати власний SQL-код до параметрів введення вебдодатку, що передаються серверній системі управління базою даних (СУБД) для аналізу та виконання [7, с. 6]. Тобто SQL-ін’єкція виникає саме в той момент, коли програма формує SQL-запит шляхом конкатенації рядка з даними, які ввів користувач за умови відсутності перевірки та екранування запиту. Так зловмисник отримує можливість змінити логіку та структуру SQL-запит, що призводить до виконання БД дій, які розробник не передбачав та не дозволяв.

За словами М. Алсами, SQL-ін’єкція – це такий метод передавання SQL-коду до вебдодатків, при якому дані користувача обробляються способом, що не відповідає первісному задуму розробника [2, с. 1]. Отже, мета такої атаки – змусити БД виконати шкідливий код, користуючись недосконалістю архітектури вебдодатку.

Розглянемо принцип роботи SQL-ін’єкції через принцип роботи типового вебдодатку, який взаємодіє з БД. Більшість вебдодатків побудовані за допомогою трирівневої архітектури: рівень представлення (браузер), логічний рівень (вебсервер) та рівень зберігання (база даних). Логічний рівень формує SQL-запити на основі введених користувачем даних та передає їх до бази даних [7, с. 3]. Проблема виникає тоді, коли розробник будує SQL-запит шляхом динамічної конкатенації рядків – безпосередньо підставляє введені користувачем дані в тіло SQL-виразу.

Розглянемо приклад такого підходу на формі автентифікації, написаній на PHP [7, с. 8]:

```
$query = "SELECT userid FROM CMSUsers WHERE user = '$_GET[user]'  
AND password = '$_GET[password]'";
```

Лістинг 1.1. Вразливий SQL-запит

Якщо зловмисник введе в поле пароля значення «bar or 1=1», запит буде виглядати такі:

```
$SELECT userid FROM CMSUsers WHERE user = 'foo' AND password =  
'bar' OR '1' = '1';
```

Лістинг 1.2. Виконання SQL-ін'єкції в запиті

Умова «or 1=1» є завжди істинною, і система повертає перший запис із таблиці користувачів. Так зловмисник отримує доступ до захищених даних, навіть не знаючи пароль [7, с. 8-9].

Н. Пател виділяє чотири основні причини виникнення вразливостей такого типу: некоректна обробка escape символів, некоректна обробка типів даних, некоректне складання запитів та неналежна обробка помилок БД [3, с. 199].

Розглянемо їх детальніше.

1. Некоректна обробка escape-символів – оскільки SQL інтерпретує одинарні лапки «'» як межу між кодом та даними, то все, що написано після лапок, розглядається як виконуваний код. Відповідно у випадку відсутності екранування ця вразливість може бути використана зловмисником для вставки власних SQL-команд [7, с. 14].

2. Некоректна обробка типів даних – у випадку передавання числових параметрів без перевірки їх типу виникає загроза підставки зловмисником замість числових даних довільного SQL-виразу. Наприклад, команда LOAD_FILE, введена замість числового ідентифікатора, може дозволити зловмиснику прочитати вміст системних файлів серверу [7, с. 15-16].

3. Некоректне складання запитів притаманне складним застосункам, де ім'я таблиці або поле заздалегідь невідоме та формується динамічно. У випадку відсутності перевірки таких параметрів програмістом, вони стають вектором атаки [7, с. 17].

4. Неналежна обробка помилок дає зловмиснику цінну інформацію: повідомлення про помилки SQL може розкрити версію СУБД, структуру таблиць тощо [7, с. 18].

Додатково можна виділити ще одну причину – динамічне формування SQL-запитів через конкатенацію рядків. Більшість вебдодатків формують свої запити саме в такий спосіб, і якщо розробник не реалізує належну валідацію вхідних даних, то будь-який запит слід уважати потенційно вразливим [3, с. 200]. Отже, саме відсутність параметризації запитів є однією з найбільш розповсюджених причин SQL-ін'єкцій у додатках.

Щоб використання SQL-ін'єкції було успішним, необхідний збіг кількох умов: наявність форм, які приймають дані від користувача; використання отриманих даних при динамічному формуванні SQL-запитів; відсутність валідації, типізації та параметризації вхідних даних [2, с. 4].

SQL-ін'єкції не можна назвати однорідними: це цілий клас атак, який охоплює різні техніки, що відрізняються за механізмом впливу на БД, способом отримання відповіді від БД та власної складності. Систематизація SQL-ін'єкцій є необхідною передумовою для розуміння потенційних векторів атак та вибору відповідних засобів захисту від них.

SQL-ін'єкції умовно можна поділити за такими критеріями: за категорією методу ін'єкції, за способом отримання відповіді від БД та за порядком виконання [2, с. 2-3]. Розглянемо їх детальніше.

За методом ін'єкцій Пател виділяє чотири категорії SQL-ін'єкцій: SQL-маніпуляція, ін'єкція коду, ін'єкція через виклик функцій та переповнення буфера [3, с. 200].

SQL-маніпуляцію можна назвати найпоширенішою категорією атак. При ній зловмисник модифікує SQL-запит шляхом зміни конструкції WHERE або додавання додаткових умов за допомогою логічних операторів AND та OR. Наприклад, зловмисник додає умову OR '1'='1'. Ця умова та умови, подібні до цієї, завжди є істинними та повертають усі рядки таблиці незалежно від інших умов запити [3, с. 200]. Дуже часто такий підхід використовується для обходу механізму автентифікації та несанкціонованого доступу до захищених розділів вебдодатку.

При ін'єкції коду зловмисник вводить нові SQL-оператори в поле вводу, замість очікуваних даних, для виконання додаткових операцій у рамках одного запиту. Подібну ін'єкцію можна реалізувати тільки тоді, коли СУБД підтримує виконання кількох SQL-операторів в одному запиті або ж коли СУБД підтримує оператори AND та OR [3, с. 200].

Ін'єкція через виклик функцій передбачає вставку вбудованих функцій або визначення користувачем функцій у вразливі SQL-запити. Ці функції можуть виконувати системні виклики, зчитувати або ж модифіковувати дані в БД або ж завдати шкоду на рівні операційної системи. Наприклад, функція LOAD_FILE може бути використана зловмисником для перегляду файлів на сервері [3, с.200].

Переповнення буфера є підвидом ін'єкції через виклик функцій. У низці СУБД деякі функції містять вразливості, які призводять до переповнення буфера при передаванні великого обсягу даних. Така атака може призвести до виконання довільного коду на стороні сервера [3, с. 200].

За способом отримання відповіді від БД можна виділити такі групи: In-Band SQL-ін'єкція, Inferential SQL-ін'єкція та Out-of-Band SQL-ін'єкція [2, с. 2].

In-Band SQL-ін'єкція є однією з найбільш розповсюджених та дуже простою в реалізації. Як для проведення атаки, так і для отримання результату зловмисник використовує один і той самий канал. У цій ін'єкції виділяють два підтипи: Error Based SQL-ін'єкція та Union Based SQL-ін'єкція. Error Based є найпоширенішою формою In-Band атак. При ній зловмисник навмисно формує некоректний запит, щоб побачити повідомлення про помилку, оскільки повідомлення може містити інформацію про структуру запиту, версію СУБД, схему БД або ж про операційну систему [2, с. 3]. Отримана в такий спосіб інформація використовується для проведення подальших ін'єкцій. Union Based надає можливість зловмиснику об'єднати результат оригінального запиту з результатом власного запиту. Така операція є можливою завдяки оператору UNION. Щоб подібна операція була успішною, необхідно, щоб обидва оператори SELECT повернули однакову кількість стовбців (типи даних та порядок стовпців

також мають збігатися) [2, с. 3]. Union Based дозволяє зловмиснику отримати довільні дані з таблиць БД безпосередньо у відповіді вебдодатку.

Inferential (Blind) SQL-ін'єкція не надає зловмиснику отримати безпосередній результат виконання запиту у відповіді додатку. Така SQL-ін'єкція надає можливість зробити висновки про вміст БД на основі поведінки додатку чи часу відповіді [2, с. 3-4]. Тут також можна виділити два підтипи: Boolean-Based Blind SQL Injection та Time-Based Blind SQL Injection. Boolean-Based Blind SQL Injection покладається на формуванні запитів, які повертають різну відповідь залежно від того, чи є умова правильною або хибною. Тобто зловмисник послідовно перебирає символи необхідних даних та щоразу отримує відповідь «так» або «ні». Унаслідок цього зловмисник може побудувати повне уявлення про вміст БД [2, с. 3]. Time-Based Blind SQL Injection використовує SQL-функції затримки (SLEEP(), WAITFOR DELAY). Зловмисник надсилає запит із такою функцією й на основі часу відповіді робить висновок про правдивість умови та методично отримує інформацію [2, с. 3].

Out-of-Band SQL-ін'єкція є найменш поширеною, оскільки вона вимагає наявності певних специфічних функцій на сервері БД. При такій ін'єкції зловмисник проводить атаку та отримує дані через різні канали зв'язку. Наприклад, зловмисник ініціює DNS або HyperText Transfer Protocol (HTTP) запит від сервера БД до підконтрольного сервера, використавши його для передавання даних [2, с. 3]. Out-of-Band використовується, коли In-Band або Blind ін'єкції є малоефективними.

За порядком виконання виділяють ін'єкції першого порядку (First-Order SQL-Injection) та другого порядку (Second-Order SQL-Injection) [2, с. 2].

При ін'єкції першого порядку шкідливий код вводиться та виконується одразу в межах одного сеансу взаємодії з додатком. Переважна більшість розглянутих вище атак відноситься до цієї групи [2, с. 2].

При ін'єкції другого порядку (відкладеній ін'єкції) шкідливий код спочатку зберігається, але одразу не виконується. За задумом зловмисника, шкідливий код пізніше буде вилучений та використаний в іншому SQL-запиті.

Подібні атаки є доволі складними для виявлення, оскільки на етапі введення даних вразливість може ніяк не проявитися [2, с. 2]. Ін'єкції другого порядку особливо небезпечні для систем, де збережені дані використовуються для формування динамічних запитів.

У сучасних, присвячених автоматизованому виявленню SQL-ін'єкцій засобами машинного навчання дослідженнях, присутні два підходи до аналізу. Перший підхід передбачає аналіз SQL-запитів як окремих рядків даних (SQL detection) та забезпечує точність детекції (до $F1=0.99$) [6, с. 2]. Другий підхід спрямований на виявлення вразливих SQL-запитів у вихідному коді вебдодатку (code-base detection), тобто в тих фрагментах коду (у тому числі й Python коду), де формуються та виконуються небезпечні SQL-вирази. Такий підхід складніший в реалізації, ніж перший, оскільки він потребує аналізу контексту виконання запиту, відстеження від вхідного параметру до SQL-виразу та дисбалансу класів у навчальних вибірках [6, с. 2].

Актуальність виявлення SQL-ін'єкцій зумовлена тим, що незважаючи на давність проблеми, SQL-ін'єкції входять до трійки найбільш небезпечних вразливостей (звіти OWASP за 2012-2021) [6, с. 2].

Отже, класифікація SQL-ін'єкцій охоплює декілька вимірів: за методом маніпуляції запитом, за механізмом отримання відповіді, за порядком виконання та за рівнем аналізу в контексті виявлення вразливостей. Розуміння класифікації є необхідним для проєктування комплексної системи захисту, яка могла б охопити всі можливі вектори атак.

1.3. Аналіз сучасних інцидентів та статистики вразливостей

Оцінка загрози SQL-ін'єкцій неможлива без звернення до статистичних даних та аналізу інцидентів. Динаміка кіберзагроз свідчить про стабільно високу актуальність вразливостей цього типу, попри тривалий розвиток технологій захисту та зростання обізнаності розробників в цьому питанні.

Згідно рейтингами OWASP Top 10 (2013 р., 2017 р.), сформованими на основі аналізу даних від організацій по всьому світу, SQL-ін'єкції посідають першу позицію категорії A1, що підтверджує їх роль серед кіберзагроз [2, с. 2].

Слоткієн також зазначає, що SQL-ін'єкції входили до трійки найнебезпечніших вразливостей у звітах OWASP Top 10 кожен рік з 2013 по 2021. Це є свідченням неефективності методів, які існують для виявлення й усунення SQL-ін'єкції [6, с. 2].

Слід зазначити, що рейтинг OWASP формується не лише за частотою виявлення, а й за оцінкою ризику (поширеність, складність виявлення та потенційний вплив на систему). Стабільне лідерство SQL-ін'єкцій в рейтингу свідчить про системний характер проблеми. SQL-ін'єкції не є наслідком специфічної платформи чи технологій: вони відображають фундаментальну помилку в розробці – недостатню увагу до валідації та параметризації вхідних даних.

За даними Verizon Data Breach Investigations Report (DBIR) 2024 р., SQL-ін'єкції та інші атаки на вебзастосунки становили 26% від усіх зафіксованих витоків даних у 2024 році [8]. За оцінками Aikido Security, у 2024 році 6,7% вразливостей, виявлених у проєктах із відкритим кодом припадали саме на SQL-ін'єкції. У проєктах із закритим кодом – 10% [9]. Загальна кількість зафіксованих SQL-ін'єкцій у 2024 році зросла порівняно з попереднім роком (2264 ін'єкції в 2023 році та 2400 у 2024 році). Слід зазначити, що відсоток SQL-ін'єкцій від всіх вразливостей дещо знизився. Також за даними Aikido Security, більше 20% комерційних проєктів при першому аудиті кібербезпеки містили мінімум одну SQL-ін'єкцію, а середня кількість вразливих місць у подібних проєктах може налічувати до 30 ділянок коду [9].

За даними WhiteHat Security, 98% вебзастосунків містять хоча б одну вразливість, а SQL-ін'єкції становлять 42% атак [8]. База даних Common Vulnerabilities and Exposures (CVE) налічує більше 15000 записів, пов'язаних з SQL-ін'єкціями [10].

Для об'єктивної оцінки критичності вразливостей використовується Common Vulnerability Scoring System (CVSS) – загальна система оцінки вразливостей. CVSS формує числову оцінку від 1 до 10 залежно від рівня

загрози, де 0 – загроза відсутня, 0,1-3,9 – низька, 4-6,9 – середня, 7-8,9 – висока, 9-10 – критична [12].

Перехід від статистичних даних до реальних кейсів дозволяє оцінити масштаб загрози. У 2020 році компанія Freepik повідомила про витік даних понад 8 мільйонів користувачів Унаслідок SQL-ін'єкції [10]. Цей витік є одним із найбільших в індустрії цифрового контенту. Він показав, що навіть великі технологічні компанії залишаються вразливими.

Ще одним резонансним інцидентом стала атака на платформу MOVEit Transfer (передача файлів), яка належить компанії Progress Software. Цим програмним забезпеченням користуються переважно урядові, медичні та фінансові організації. 27 травня 2023 року група CL0P (TA505) почала експлуатувати критичну SQL-ін'єкцію нульового дня CVE-2023-34362 (оцінка небезпеки CVSS 9,8). Після успішної SQL-ін'єкції зловмисники встановили власний webshell LEMURLOOT для закріплення в системі та подальшого викрадення даних [11].

У листопаді-грудні 2023 року група ResumeLooters, використовуючи переважно SQL-ін'єкції, провела масову атаку на понад 65 сайтів у Тихоокеанському регіоні. Було викрадено понад двох мільйонів записів із персональними даними користувачів. Згодом ці дані були виставлені на продаж [13].

Окремо треба видалити новий вектор загроз – SQL-ін'єкції в коді, згенерованому штучним інтелектом (ШІ). Великі мовні моделі (LLM) вносять SQL-ін'єкції в згенерований ними код [6, с. 2]. Отже, зростання популярності штучного інтелекту в сфері розроблення програмного забезпечення не лише не вирішує її, але й є одним із факторів її поширення, особливо серед недосвідчених розробників, які не перевіряють згенерований код. І.Замрій та І.Шахматов наголошують, що формування якісних навчальних вибірок суттєво покращують точність автоматичного виявлення вразливостей [5, с. 60].

Незважаючи на наявність широкого спектру автоматизованих засобів виявлення SQL-ін'єкцій, їх ефективність залишається доволі обмеженою.

Наприклад, А. Федоренко встановив, що жоден із популярних сканерів (Acunetix, Burp Suite, Nessus, OpenVas, SQLMap тощо) не здатен сам по собі забезпечити достатній захист. Одна частина цих сканерів не в змозі виявити складні SQL-ін'єкції, інша частина – генерує результати, які вводять в оману [4, с. 59-61]. Порівняльне дослідження, яке провели І. Замрій та І. Шахматов на платформі WAVSEP показало суттєві відмінності в роботі сканерів: Acunetix, SkipFish та ZAP показали найвище значення F1-міри, тоді як W3AF показав низьку точність [5, с. 62].

Тож, аналіз сучасної статистики та реальних кейсів показує, що SQL-ін'єкції залишаються однією з найнебезпечніших та найпоширеніших загроз. Аналіз реальних кейсів показує, що незважаючи на те, що SQL-ін'єкції вже широко відомі, вони продовжують завдавати доволі великої шкоди (репутаційної, фінансової) як різним компаніям, так і їх клієнтам. Лідерство в рейтингах OWASP та поява нового вектора через код, згенерований штучним інтелектом, зумовлює необхідність системного підходу до захисту вебзастосунків від SQL-ін'єкцій.

Висновки до розділу 1

У розділі 1 здійснено аналіз проблеми SQL-ін'єкцій у веб-додатках: досліджено загальний стан безпеки вебзастосунків, визначено поняття та класифікацію SQL-ін'єкцій, а проаналізовано сучасну статистику вразливостей і реальні інциденти.

Аналіз звітів провідних організацій кібербезпеки (OWASP, Veracode, Verizon DBIR) підтвердив, що SQL-ін'єкції стабільно входять до трійки найнебезпечніших вразливостей упродовж 2013-2021 років та посідають позицію A03 у рейтингу OWASP Top 10 (2021). За даними Verizon DBIR 2024, атаки на вебзастосунки становили 26% від усіх зафіксованих витоків даних, а база CVE налічує понад 15000 записів, пов'язаних із SQL-ін'єкціями. Це свідчить про системний, а не ситуативний характер загрози.

Установлено, що SQL-ін'єкції виникають у момент, коли програма формує SQL-запит через конкатенацію рядків без належної перевірки й екранування

введених користувачем даних. Виявлено чотири основні технічні причини виникнення вразливостей, як-от: некоректна обробка escape-символів, некоректна обробка типів даних, динамічне складання запитів і неналежна обробка помилок бази даних.

Класифіковано SQL-ін'єкції за трьома вимірами: за методом ін'єкції (SQL-маніпуляція, ін'єкція коду, виклик функцій, переповнення буфера), за способом отримання відповіді (In-Band, Inferential/Blind, Out-of-Band) та за порядком виконання (першого та другого порядку). Особливу небезпеку становлять ін'єкції другого порядку (Second-Order Injection), оскільки шкідливий код спочатку зберігається в базі даних і лише згодом виконується, що суттєво ускладнює їх виявлення.

Аналіз реальних інцидентів підтвердив критичну небезпеку SQL-ін'єкцій: атака на платформу MOVEit Transfer (CVE-2023-34362, CVSS 9,8) у 2023 році призвела до компрометації урядових та фінансових організацій; масова атака групи ResumeLooters на 65 вебсайтів спричинила викрадення понад двох мільйонів записів із персональними даними. Також виявлено новий вектор загроз – SQL-ін'єкції в коді, згенерованому LLM, що актуалізує проблему для сучасних проєктів із застосуванням штучного інтелекту.

Установлено, що використання жодного з автоматизованих сканерів безпеки (Acunetix, Burp Suite, SQLMap тощо) як єдиного засобу захисту є недостатнім, оскільки частина з них не виявляє складних SQL-ін'єкцій, а інша генерує хибно-позитивні результати. Комплексний підхід, що поєднує автоматизоване та ручне тестування й практики DevSecOps, є єдиним надійним способом виявлення та усунення вразливостей.

РОЗДІЛ 2

МЕТОДИ ТА ЗАСОБИ ЗАХИСТУ ВІД SQL-ІН'ЄКЦІЙ

2.1. Загальні принципи безпечної розробки веб-додатків

Без дотримання принципів безпечної розробки вебдодатків неможливо реалізувати захист від SQL-ін'єкцій. Ці принципи формують концептуальну основу, на якій, власне, і базуються технічні механізми захисту. Порушення цих принципів є однією з причин виникнення вразливостей у програмному забезпеченні. Безпека вебдодатків – це наслідок системного підходу, який охоплює всі етапи розроблення: проектування, реалізацію, тестування та супроводження [5, с. 59].

Одним із ключових принципів можна назвати принцип мінімізації привілеїв (Principle of Least Privilege). Згідно з цим принципом, кожен компонент системи, що використовується вебдодатком, отримує мінімальний набір дозволів, необхідних для його роботи [7, с. 21]. Наприклад, якщо обліковий запис бази даних у додатку має право на операції SELECT, UPDATE, INSERT стосовно певних таблиць, то навіть у випадку успішної SQL-ін'єкції злоумисник не зможе видалити дані, отримати права адміністратора або паралізувати роботу СУБД. Обліковий запис не має мати права на операції DROP, CREATE, ALTER або ж доступу до системних таблиць [12, с. 69].

Іншим важливим принципом є принцип ешелонованого захисту (Defense in Depth). Згідно з цим принципом, ні в якому разі не можна будувати безпеку додатку, покладаючись лише на один механізм захисту. У системи має бути декілька незалежних рівнів захисту. Подолання одного з рівнів не має дати злоумиснику можливість повністю скомпрометувати систему. У контексті SQL-ін'єкцій це означає поєднання валідації вхідних даних, параметризації запитів, обмеження привілеїв та логування [5, с. 59].

Принцип безпеки за замовчуванням (Secure by Default) вимагає налаштування системи в найбезпечнішому режимі з самого початку. Тобто всі вхідні дані вважаються ненадійними та такими, які можуть нанести шкоду, а

доступ до ресурсів надається лише тільки тоді, коли є явні підстави для цього. Динамічне формування SQL-запитів шляхом конкатенації рядків з введеними користувачем даними є неприпустимим [7, с. 22].

За принципом мінімальної поверхні атаки (Attack Surface Reduction) необхідно максимально обмежити кількість вхідних точок, через які зловмисник може взаємодіяти з БД. Тобто необхідно реалізувати таке: відключити непотрібні функції СУБД; обмежити перелік таблиць і полів, доступних через публічний API; використовувати збережені процедури тільки там, де це продиктовано логікою застосунку [7, с. 21-22]. Зменшення вхідних точок неодмінно призводить до зменшення вірогідності успішного проведення атаки.

Важливе місце також посідає принцип розмежування відповідальності (Separation of Concerns). Порушення цього принципу через конкатенацію SQL-рядків з введеними користувачем даними є технічною першопричиною SQL-ін'єкцій [7, с. 13]. Цей поділ реалізують параметризовані запити, оскільки вони передають SQL-код та дані до СУБД окремими каналами. Це дозволяє СУБД інтерпретувати передані значення не як частину SQL-виразу, а власне як дані.

Ключовим принципом можна назвати принцип повної відмови від довіри до вхідних даних (No Trust User Input). Згідно з цим принципом, будь-які дані, що надходять від користувача через будь-який канал введення (форми, параметри Uniform Resource Locator (URL), HTTP-заголовки, cookie-файли тощо) незалежно від контексту слід розглядати як потенційно небезпечні [14, с. 484]. Жодні дані не повинні бути включеними в SQL-запити без належної обробки – параметризації, екранування, перевірки типів. Тут є особливо небезпечними ін'єкції другого порядку (Second-Order Injection), коли шкідливий код не діє одразу, а спочатку зберігається в БД, а потім використовується при формуванні нового SQL-запиту [2, с. 2]. У таких випадках необхідне використання параметризації на всіх рівнях формування запитів, оскільки перевірки введених користувачем даних недостатньо.

Окремо слід розглянути обробку помилок. Детальні повідомлення про помилки в БД, з точки зору безпеки, не повинні відображатися для користувача.

Подібні повідомлення можуть дати зловмиснику інформацію про БД: структуру таблиць, версію СУБД, схему БД, що спростить для нього подальші атаки [7, с. 18]. Власне, на цьому й ґрунтується цілий клас атак – Error-Based SQL-Injection, при якому зловмисник цілеспрямовано формує некоректний запит, щоб отримати інформацію через повідомлення про помилку [2, с. 3]. Коректна обробка помилок передбачає такі кроки: логування, перехоплення виключень на рівні застосунку, повідомлення користувачеві лише загальної інформації про помилку без розкриття деталей.

У площині облікових записів БД необхідно дотримуватися таких вимог: для підключення вебдодатку до СУБД необхідно використовувати спеціально виділений обліковий запис, відмінний від облікового запису адміністратора БД [12, с. 69]; паролі підключення слід зберігати в змінних середовищах або в захищених сховищах секретів; будь-які чутливі конфігураційні параметри повинні бути виключені з системи контролю версій.

Важливим є також регулярний аудит та тестування безпеки в процесі розроблення. Статичний аналіз коду дає можливість виявити потенційно вразливі місця ще під час розробки, а тестування на проникнення разом із динамічним тестуванням дають змогу перевірити захист безпосередньо в середовищі виконання [14, с. 484]. Інтеграція цих практик у цикл DevSecOps здатна забезпечити систематичне виявлення вразливостей ще на ранніх стадіях. За результатами досліджень, ручна перевірка в поєднанні з автоматизованим скануванням дає кращі результати, ніж кожен підхід окремо [4, с. 7].

Отже, принципи безпечної розробки (мінімізація привілеїв, ешелонований захист, безпека за замовчуванням, відмова довіри до вхідних даних, мінімальна поверхня атаки) створюють основу, на якій базується захист вебдодатку від SQL-ін'єкцій.

2.2. Підготовлені запити та ORM

Підготовлені запити (preparing statements) та об'єктно-реляційне відображення (Object-Relational Mapping, ORM) – фундаментальні технічні механізми захисту від SQL-ін'єкцій. Як перший, так і другий підходи реалізують

на практиці принцип розмежування SQL-коду та даних користувачів, але відбувається це на різних рівнях абстракції.

Підготовленим запитом називають попередньо скомпільований SQL-вираз, у якому місця для підстановки змінних значень позначаються спеціальними плейсхолдерами. Головна відмінність підготовленого запиту від динамічного полягає в тому, що структура SQL-виразу фіксується та компілюється СУБД до того, як до неї передадуться дані. Коли дані передані, СУБД інтерпретує їх виключно як значення параметрів, а не як частину SQL-коду [3, с. 201].

Отже, якщо зломисник спробує передати через поле введення рядок, наприклад, 'OR '1'='1, СУБД обробить його як звичайний текстовий рядок, а не як SQL-вираз.

Технічна реалізація підготовленого запиту є двоетапною.

Під час виконання першого етапу застосунок надсилає СУБД шаблон SQL-запиту з плейсхолдерами (%s, \$1 тощо), СУБД у свою чергу його аналізує, перевіряє та складає план виконання.

Під час другого етапу застосунок передає значення параметрів окремим каналом, а СУБД підставляє їх у готовий план виконання [7, с. 13].

Логіка або структура SQL-запиту не може бути змінена зломисником через те, що структура SQL-запиту була зафіксована ще на першому етапі.

У Python для роботи з PostgreSQL на низькому рівні використовується бібліотека `psycopg2`. Вона реалізує специфікацію DB-API 2.0 та підтримує параметризовані запити.

Розглянемо приклад використання параметризованого запиту в `psycopg2` за допомогою табл. 2.1.

Таблиця 2.1

Використання параметризованого запиту в `psycopg2`

Запит з використанням конкатенації рядків	Параметризований запит <code>psycopg2</code>
---	--

Продовження табл. 2.1

<pre>username = request.POST['username'] query = "SELECT * FROM users WHERE username = ' " + username + "'" cursor.execute(query)</pre>	<pre>username = request.POST['username'] cursor.execute("SELECT * FROM users WHERE username = %s", (username,))</pre>
---	---

У прикладі значення `username` передається в якості окремого параметра, а не як частина рядку запиту. Екранування переданих значень відповідно до вимог PostgreSQL, бібліотека `psycopg2` виконує самостійно, що виключає можливість SQL-ін'єкції [12, с. 69]. Тут важливо використовувати саме кортеж, або список в якості другого аргументу методу `execute()`, а не підставляти параметри за допомогою форматування рядків Python, оскільки це приводить до створення вразливого запиту.

Ще однією перевагою підготовлених запитів є використання СУБД уже складеного плану виконання, якщо використовується один і той самий шаблон із різними параметрами, що є плюсом для продуктивності роботи [3, с. 201].

ORM є більш високорівневим підходом, бо не тільки вирішує проблему SQL-ін'єкцій, але й ще спрощує взаємодію користувача з БД через застосунок. ORM забезпечує відображення між об'єктами застосунку та таблицями БД, тим самим даючи розробнику можливість не писати SQL-запити, а працювати з даними через об'єктно-орієнтований API [5, с. 57].

Django ORM є одним із найбільш функціональних ORM в екосистемі Python. Його основою є класи моделей – підкласи `django.db.models.Model`, де кожне поле класу відповідає атрибуту сутності в БД. При виконанні операцій з даними через API Django ORM генерує параметризовані запити та передає їх до СУБД. Це виключає безпосереднє включення даних користувача в тіло SQL-виразу [2, с. 5].

Розглянемо операції з даними в Django ORM за допомогою узагальнювальної табл. 2.2.

Таблиця 2.2

Операції з даними в Django ORM

Дія	Метод Django ORM	SQL-еквівалент	Код
Отримання запису	.get()	Параметризований SELECT	<code>user = User.objects.get(username=request.POST['username'])</code>
Фільтрація	.filter()	Параметризований WHERE	<code>lessons = Lesson.objects.filter(semester=semester_id, week=week_number)</code>
Створення запису	.create()	Параметризований INSERT	<code>Lesson.objects.create(title=title, teacher=teacher, room=room)</code>
Оновлення	.update()	Параметризований UPDATE	<code>Lesson.objects.filter(id=lesson_id).update(title=new_title)</code>

У наведених прикладах значення, які передали як аргументи методів ORM, обробляються не як частина SQL-коду, а як параметри запитів. Django ORM передає ці виклики в параметризовані запити через `psycopg2` при роботі з PostgreSQL, що дає захист від SQL-ін'єкцій на архітектурному рівні.

Також у Django ORM є механізм `QuerySet` – відкладене виконання запитів. Це робить можливим поступове створення складних запитів через об'єднання умов фільтрації, сортування та анотацій. Запит виконується тоді, коли `QuerySet` обчислюється, наприклад, при ітерації, перетворенні у список або передачі в шаблон, тобто тоді, коли це потрібно [15]. Кожен виклик методів `filter()`, `exclude()`, `order_by()` тощо повертає новий `QuerySet` при збереженні попереднього. Це надає можливість будувати запити поступово й безпечно передавати їх між функціями [15].

Незважаючи на всі переваги Django ORM, можливе виникнення ситуацій, коли необхідне виконання складних запитів, які не можуть бути виражені засобами стандартного ORM API. У таких ситуаціях використовуються метод `raw()` та інтерфейс `cursor()`. Використання `raw()` дозволяє виконувати довільні SQL-запити, повертаючи об'єкти моделі. Але при використанні `raw()` необхідно дотримуватися параметризації. Це відноситься й до `cursor()`. При будь-якому

зверненні до СУБД усі змінні значення мають передаватися тільки як параметри й у жодному разі не включатися в рядок запиту шляхом форматування [7, с. 14].

Робота Django ORM із динамічними іменами полів і таблиць має певну особливість: методи ORM приймають імена полів як рядкові аргументи, але імена не екрануються так само як параметри значень, а порівнюються з переліком відомих полів моделі. Якщо ім'я поля передається динамічно на основі введених даних, необхідно явно перевірити його приналежність до допустимого переліку полів до передачі в метод ORM. Це необхідно зробити для уникнення можливої маніпуляції структурою даних [2, с. 5].

2.3. Валідація та санітизація вхідних даних

Важливим доповненням до ORM та параметризованих запитів є валідація та санітизація даних, оскільки вони формують додатковий рівень захисту за принципом ешелонованого захисту. Параметризовані запити можна вважати вже достатнім захистом від SQL-ін'єкцій на рівні БД, але слід пам'ятати, що валідація вхідних даних виявляє та блокує некоректні та потенційно шкідливі дані до того моменту, коли вони досягають рівня формування запиту, що звужує загальну поверхню атаки [14, с. 484].

Необхідно розмежувати два визначення – валідацію та санітизацію. Нерідко вони вживаються як синоніми, але мають різне значення. Валідація – перевірка, чи відповідають вхідні дані очікуваному формату, типу, діапазону значень або іншим уже визначеним правилам. У випадку, коли дані не відповідають визначеним правилам, вони відхиляються, а користувач отримує повідомлення про помилку. Санітизація ж є перетворенням вхідних даних у безпечний вигляд через видалення, екранування або заміну потенційно небезпечних символів або конструкцій [3, с. 202]. Підходи взаємодоповнюючі – валідація відхиляє неприйнятні дані, а санітизація перетворює потенційно небезпечні дані безпечними для подальшої їх обробки.

Валідація вхідних даних має здійснюватися одразу на двох рівнях – клієнтському та серверному. Валідацію на стороні клієнта ні в якому разі не можна вважати надійною, оскільки зловмисник може її обійти через надсилання

HTTP-запитів в обхід браузера. Але все ж таки клієнтська валідація знижує навантаження на сервер та надає користувачу миттєвий зворотній зв'язок [Кларк, с. 22]. Тільки серверну валідацію можна назвати реальним захистом від злоумисників, саме тому валідація на стороні сервера має бути в незалежності від наявності валідації на стороні клієнта [12, с. 70].

У Django основним механізмом валідації на стороні сервера є Django Forms. Клас форми описує поля, їх типи та правила валідації, а метод `is_valid()` запускає процес санітизації та валідації вхідних даних [16]. Очищені та нормалізовані дані доступні через атрибут `cleaned_data`, який є словником, у якому ключі – імена полів, а значення – Python-об'єкти (перевірені та приведені до правильного типу).

Django Forms використовує декілька рівнів обробки для кожного поля. Першим кроком є використання методу `to_python()`, який приводить сирі дані з HTTP запити до правильного типу даних у Python (наприклад, `FloatField` переводить дані в число з плаваючою помилкою), або ж, якщо перетворення неможливе, він повертає `ValidationError`. Після чого метод `validate` перевіряє на відповідність специфічним правилам поля, а `run_validators` послідовно запускає всі прив'язані до поля валідатори та об'єднує всі помилки в один `ValidationError` [16]. Унаслідок використання такого підходу до `cleaned_data` потрапляють тільки коректні та типізовані дані.

У контексті захисту від SQL-ін'єкцій одним із найважливіших аспектів валідації можна назвати типізацію. Наприклад, якщо поле оголошене як `IntegerField`, то в `cleaned_data` потрапить тільки ціле число. Будь-які спроби передати рядок з SQL-кодом будуть відхилені ще на етапі `to_python()` [16].

Для більшого контролю над вхідними значеннями в Django існують callable-об'єкти, які приймають значення та генерують `ValidationError` у випадку, коли значення не відповідають заданим правилам, а вбудований клас `RegexValidator` перевіряє, чи відповідає значення регулярному виразу [17]. Розглянемо приклад використання регулярного виразу для поля введення імені користувача (дозволені тільки літери та цифри):


```

from django.core.validators import RegexValidator
from django import forms

username_validator = RegexValidator(
    regex=r'^[a-zA-Z0-9a-яA-ЯієrІіЄГ]+$',
    message= "Ім'я користувача може містити тільки літери та цифри"
)

class UserForm (forms.Form):
    username = forms.CharField(
        max_length = 50,
        validators = [username_validator]
    )

```

Лістинг 2.1. Валідація поля введення за допомогою RegexValidator

Цей підхід забезпечує валідацію за принципом «білого списку», тобто, замість спроб виявити та знешкодити всі можливі шкідливі конструкції, перевіряється, чи належать значення до вже визначеної множини дозволених символів [12, с. 24]. Від повноти заблокованих конструкцій «білий список» ніяким чином не залежить, тому він є надійнішим за «чорний список».

Розглянемо приклад для полів з обмеженим переліком допустимих значень з використанням вибіркового поля (ChoiceField), де форма обмежує вхідні дані переліком заздалегідь визначених варіантів:

```

class LessonFilterForm(forms.Form):
    LESSON_TYPES = [
        ('lecture ', 'Лекція'),
        ('practice ', 'Практика'),
        ('lab ', 'Лабораторна'),
    ]
    lesson_type = forms.ChoiceField(choices= LESSON_TYPES)
    week_number = forms.IntegerField(min_value=1,
max_value=20)

```

Лістинг 2.2. Обмеження вхідних даних за допомогою ChoiceField та IntegerField

Поле `lesson_type` приймає значення тільки з визначеного переліку, а `week_number` – тільки цілі числа в діапазоні від 1 до 20. Усі значення, відмінні від зазначених вище, відсіюються ще до того, як потраплять в ORM-запит. Цей підхід реалізовує принцип предметно-орієнтованого обмеження вхідних даних (Taking User Input From Predefined Choices) [3, с. 202].

Окремо слід виділити довжину полів, оскільки відсутність обмеження на максимальну довжину рядка є вразливістю, оскільки зловмисник може використати її для атаки, які пов'язані з переповненням або надмірним споживанням ресурсів. Django CharField приймає атрибут `max_length` (який є обов'язковим), який і встановлює обмеження на довжину рядка [16]. Ця перевірка здійснюється і на рівні сервера, і на рівні клієнта.

Також важливою складовою є валідація на рівні моделі Django. Поля моделі мають власні обмеження (`max_length`, `null`, `blank`, `unique`). При збереженні об'єкта через ModelForm валідатори моделі запускаються автоматично [16]. Це формує додатковий рівень захисту – некоректні значення відхиляються на рівні моделі.

Слід також зазначити, що при використанні ORM та параметризованих запитів екранування небезпечних SQL-символів виконується на рівні драйвера БД, тобто потреба в ручному екрануванні зникає. Але екранування залишається необхідним при виводі даних назад у HyperText Markup Language (HTML) для запобігання XSS-атакам. Тут шаблонізатор Django виконує екранування за замовчуванням [18]. Це унеможливорює введення шкідливого коду через поля введення.

Отже, валідація та санітизація в Django-застосунку реалізована через систему форм. Це забезпечує типізацію даних, а також перевірку діапазонів та форматів, нормалізацію значень та валідацію за принципом білого списку. Разом із параметризованими запитами та ORM і механізми формують багаторівневий захист від SQL-ін'єкцій.

2.4. Механізми захисту у фреймворці Django

Django – це фреймворк, який був розроблений з увагою до питань, які стосуються безпеки. Документація Django ставить безпеку в перший ряд при розробці вебдодатків. Вона зазначає, що фреймворк дає розробникам велику кількість вбудованих засобів захисту проти найпоширеніших векторів атак [18].

Центральний механізм захисту від SQL-ін'єкцій у Django – це QuerySet API. Під час побудови SQL-запитів він автоматично застосовує параметризацію. QuerySet має захист від SQL-ін'єкцій, оскільки запити будуються з використанням параметризації: SQL-код запиту визначається окремо від його параметрів, самі ж параметри екрануються драйвером БД [18]. Тобто при стандартному використанні ORM розробник отримує захист автоматично.

Також Django надає інструменти для виконання сирих SQL-запитів: конструкцію RawSQL та методи raw() та extra(). Але їх слід використовувати обережно, оскільки в рамках використання цих інструментів розробник особисто несе відповідальність за коректне екранування [18].

Важливою складовою системи захисту Django можна назвати систему middleware, яка обробляє HTTP-запити та відповіді до того, як вони досягнуть view або ж повернуться клієнту. Наскрізні аспекти безпеки middleware реалізує на рівні всього додатку [19]. Django включає до конфігурації middleware кілька класів, які відносяться до безпеки.

SecurityMiddleware реалізує налаштування безпеки на рівні HTTP-заголовків відповідей. Наприклад, він може примусово перенаправляти HTTP-запити HyperText Transfer Protocol Secure (HTTPS), установити заголовок Strict Transport Security (HSTS) (в якості захисту від атак типу downgrade та встановити параметр X-Content-Type-Options: nosniff як захист від MIME-sniffing атак і Refferer-Policy, щоб контролювати передачу реферера [19]. Дані налаштування безпосередньо не дають захисту від SQL-ін'єкцій, але ігнорувати їх не слід через те, що вони є частиною комплексного захисту вебзастосунку від суміжних загроз.

CsrfViewMiddleware (CSRF) надає захист від міжсайтової підробки запиту. Прямого зв'язку між SQL-ін'єкціями та CSRF немає, лише опосередкований, але ігнорування його ставить під загрозу безпеку вебдодатку. Наприклад, успішна CSRF-атака може призвести до виконання автентифікованим користувачем шкідливого запиту, що спричинює несанкціоновані операції з БД. Механізм захисту від CSRF-атак реалізований через перевірку секретного токена в кожному POST-запиті [18]. Генерування токена відбувається під час завантаження сторінки, зберігається він у файлах cookie та прихованому полі форми та перевіряється сервером при отриманні запиту. Зловмисник з іншого домену не може підробити запит, оскільки не має доступу до токена. В Django-шаблоні токен можна встановити за допомогою тегу `{%csrf_token%}` в тезі `<form>`.

AuthenticationMiddleware та SessionMiddleware реалізують систему автентифікації та управління сесіями. Вони важливі з точки зору БД, оскільки забезпечують розмежування доступу – автентифікований користувач отримує тільки ті права, які відповідають його ролі, що реалізує принцип мінімізації привілеїв на рівні застосунку [18].

Система автентифікації Django надає готову реалізацію механізмів входу, виходу й управління паролями. Паролі зберігаються в БД тільки в хешованому вигляді з використанням алгоритму PBKDF2 [18]. Тобто в разі витоку даних із таблиці користувачів зловмисник не отримає паролі у відкритому вигляді. Для розмежування доступу до view Django надає декоратор `@login_required` та систему дозволів (permissions). Це дає можливість декларативно обмежити доступ до конкретних функцій застосунку.

Розглянемо практичну реалізацію розмежування доступу.

```
from django.contrib.auth.decorators import login_required
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView
# Для функціональних представлень
@login_required
def schedule_view(request):
    # доступно лише автентифікованим користувачам
```

```

lessons = Lesson.objects.filter(
    semester__is_active=True
).select_related('teacher', 'classroom')
return render(request, 'schedule/list.html', {'lessons':
lessons})
# Для класових представлень
class LessonListView(LoginRequiredMixin, ListView):
    model = Lesson
    template_name = 'schedule/list.html'

```

Лістинг 2.3. Розмежування доступу до представлень за допомогою

@login_required та LoginRequiredMixin

Декоратор `@login_required` автоматично переспрямовує неавтентифікованих користувачів на сторінку входу, а `LoginRequiredMixin` реалізує аналогічну поведінку для класових представлень. Отже, доступ до можливості надсилати запити до БД отримують тільки автентифіковані користувачі з необхідними правами.

Django Template Language (DTL) – ще один вбудований механізм безпеки. DTL автоматично екранує всі змінні, що виводяться у шаблоні, замінюючи потенційно небезпечні HTML-символи (<, >, &, ', ") відповідними HTML-сутностями [18]. Це унеможливорює атаки типу XSS через поля, що відображають дані з БД. Незважаючи на те, що XSS є окремим від SQL-ін'єкцій вектором атаки, автоматичне екранування шаблонізатора є важливою складовою комплексного захисту.

System Check Framework дає можливість виявити потенційні проблеми конфігурації ще до розгортання застосунку. Команда `manage.py check --deploy` виконує перевірку налаштувань production-середовища, та в разі виявлення небезпечних конфігурацій (DEBUG встановлено у True, вимкнено CSRF-middleware, відсутні налаштування HTTPS), видає попередження [19]. Це дозволяє розробнику проактивно знешкоджувати потенційні вразливості.

Важливим аспектом налаштування безпечного Django-застосунку є конфігурація файлу `settings.py` для production-середовища. Ключові параметри безпеки включають таке: `DEBUG = False`, який вимикає відображення детальних

повідомлень про помилки; SECRET_KEY, який зберігається в змінних середовища, а не в коді; перелік дозволених доменів ALLOWED_HOSTS, що дозволяє запобігти атакам через підроблений заголовок Host, та налаштування бази даних з використанням облікового запису з обмеженими правами [5, с. 59]. Усі ці параметри формують безпечне production-середовище й доповнюють вбудовані механізми захисту на рівні конфігурації.

Отже, фреймворк Django дає розробнику цілісну систему вбудованих механізмів безпеки: захист від SQL-ін'єкцій через ORM та параметризацію запитів, захист від CSRF через middleware та токени, розмежування доступу через систему автентифікації та дозволів, захист від XSS через автоматичне екранування шаблонізатора, а також інструменти для безпечної конфігурації production-середовища. Комплексне використання цих механізмів у поєднанні з описаними у попередніх підрозділах практиками (параметризованими запитами, валідацією форм та загальними принципами безпечної розробки) формує надійний багаторівневий захист вебдодатку від SQL-ін'єкцій та інших загроз.

Висновки до розділу 2

У розділі 2 систематизовано методи та засоби захисту від SQL-ін'єкцій: розглянуто принципи безпечної розробки, підготовлені запити та ORM, механізми валідації вхідних даних, а також вбудовані засоби безпеки фреймворку Django.

Визначено шість ключових принципів безпечної розробки вебзастосунків: мінімізація привілеїв (Principle of Least Privilege), ешелонований захист (Defense in Depth), безпека за замовчуванням (Secure by Default), відмова від довіри до вхідних даних (No Trust User Input), мінімізація поверхні атаки (Attack Surface Reduction) та розмежування відповідальності (Separation of Concerns). Порушення будь-якого з цих принципів є системною причиною виникнення вразливостей і не може бути компенсоване окремими технічними засобами.

Досліджено механізм підготовлених запитів (prepared statements) та об'єктно-реляційного відображення (ORM) як фундаментальних засобів захисту від SQL-ін'єкцій. Установлено, що підготовлені запити фіксують структуру

SQL-виразу до передачі даних, завдяки чому СУБД інтерпретує введені користувачем значення виключно як параметри, а не як частину SQL-коду. Django ORM автоматично генерує параметризовані запити для всіх стандартних операцій (`filter()`, `get()`, `create()`, `update()`), що забезпечує захист на архітектурному рівні. Виявлено особливість ORM щодо динамічних імен полів: вони не параметризуються, тому потребують явної перевірки через `whitelist`.

Проаналізовано систему валідації та санітизації вхідних даних у Django. Показано, що Django Forms реалізує багаторівневу обробку кожного поля: приведення до типу (`to_python()`), перевірку специфічних правил (`validate()`) та послідовний запуск валідаторів (`run_validators()`). Підкреслено перевагу підходу «білого списку» над «чорним списком» — `RegexValidator` обмежує вхідні дані допустимим набором символів незалежно від переліку відомих шкідливих конструкцій.

Систематизовано вбудовані механізми безпеки Django, що формують багаторівневий захист: CSRF-захист через `CsrfViewMiddleware`, автентифікація та авторизація (`LoginRequiredMixin`, `permission decorators`), захист від Clickjacking (`X-Frame-Options: DENY`), параметри сесійної безпеки (`SESSION_COOKIE_HTTPONLY`, `SESSION_COOKIE_SAMESITE`), підтримка HTTPS/HSTS та система журналювання подій безпеки. Установлено, що жоден окремий механізм не усуває весь клас загроз, а ефективним є лише їх комплексне застосування відповідно до принципу ешелонованого захисту.

РОЗДІЛ 3

ПРОЄКТУВАННЯ ТА СТВОРЕННЯ ВЕБДОДАТКУ

3.1. Вибір технологічного стеку та архітектури вебдодатку

Реалізація описаних у технічному завданні вимог потребує обґрунтованого вибору технологічного стеку та архітектурного підходу, які можуть забезпечити безпечну обробку даних, масштабованість системи та зручність подальшої підтримки.

Під час розробки вебдодатку управління розкладом для навчальних закладів важливим завданням є вибір надійного, безпечного та масштабованого технологічного стеку. Оскільки тематика кваліфікаційної роботи пов'язана з захистом від SQL-ін'єкцій, ключовим критерієм при виборі технологій стали наявність вбудованих механізмів безпеки та підтримка сучасних стандартів розробки вебдодатків.

Python обраний у якості основної мови програмування, що зумовлено його зрозумілим синтаксисом, наявністю розвиненої екосистеми бібліотек як для створення вебдодатків, так і для забезпечення їх безпеки. Python використовується як в навчальних, так і в промислових проєктах, підтримує принципи чистого коду та інтегрується із сучасними засобами тестування й розгортання.

Для реалізації серверної частини вебдодатку обрано фреймворк Django, який здатен надати повнофункціональне середовище для розробки безпечного вебдодатку. Даний фреймворк містить вбудовану систему автентифікації та авторизації користувачів, механізми керування сесіями та розмежування доступу. У контексті даної роботи особливе значення має ORM Django, який формує параметризовані SQL-запити та запобігає безпосередньому включенню користувацьких даних у SQL-запит, чим знижує загрозу SQL-ін'єкцій.

У якості СУБД обрано PostgreSQL. PostgreSQL здатна забезпечити високу надійність, транзакційність, підтримує складні запити та ефективну роботу з

великими обсягами даних. На етапі розробки для спрощення розгортання середовища можливе використання SQLite.

Для реалізації клієнтської частини використовуються HTML5 (структура сторінок), CSS3 (стилізація інтерфейсу) та JavaScript (інтерактивність), що гарантує кросплатформеність, коректне відображення в сучасних браузерах та можливість подальшого розширення функціоналу інтерфейсу.

Вебдодаток побудовано на основі клієнт-серверної архітектури. Клієнтська частина відповідає за відображення інтерфейсу користувача та передавання запитів. Серверна частина відповідає за обробку запитів, бізнес-логіку та взаємодію з базою даних. У серверній частині використовується архітектура Model-View-Template (MVT), що є стандартом для Django. MVT передбачає розподіл функціоналу додатку на такі компоненти:

1. Model – опис структури даних та правила взаємодії з БД.
2. View – реалізація бізнес-логіки та обробка HTTP запитів.
3. Template – формування користувацького інтерфейсу.

Дане архітектурне рішення сприяє підвищенню підтримуваності коду, зменшує ризик помилок, що пов'язані з некоректною обробкою даних та спрощує тестування. Чіткий розподіл відповідальностей між компонентами дає можливість централізовано реалізувати механізми захисту від SQL-ін'єкцій на рівні моделей та представлень.

У рамках виконання кваліфікаційної роботи було спроектовано структуру майбутнього додатку та визначено взаємозв'язки між компонентами системи. Результатом роботи на цьому етапі є структурна схема, яка відображає трьохшарову архітектурну організацію системи: 1 – клієнтський, 2 – серверний та 3 – шар даних. Структурна схема необхідна для визначення як складу, так і взаємодії компонентів системи ще до реалізації самого додатку. Вона є основою для подальшого прийняття архітектурних рішень.

Клієнтський шар реалізований за допомогою HTML, CSS та Javascript. Такий підхід обумовлений специфікою архітектури MVT (формування розмітки сторінок відбувається за допомогою шаблонізатора Django) на стороні сервера.

Клієнтська частина відповідає тільки за відображення HTML у браузері та забезпечення інтерактивності за допомогою Javascript. Взаємодія клієнтського шару та сервера відбувається через HTTP-протокол: браузер надсилає GET та POST запити, сервер у свою чергу їх обробляє та повертає назад уже відрендерену HTML-сторінку. Усі форми клієнтської частини мають CSRF-токен. Даний токен автоматично генерується самим Django та перевіряється сервером, коли той перевіряє POST-запити. Такий підхід унеможливорює виконання міжсайтових підроблених запитів, які використовуються як вектор для атак на базу даних. Користувачів, які взаємодіють із системою, можна розділити на три групи: викладачі, адміністратори та студенти. Кожна група отримує доступ виключно до тих сторінок, які відповідають визначеній ролі. Розмежування доступу реалізовано на рівні представлень Django декоратором `@login_required` та перевірки ролі у відповідних View-класах.

Центральним компонентом в архітектурі є серверний шар. Відповідно до шаблону MVT, шар поділений на три рівні: Templates, Views, Models. У кожного рівня є свої визначені функції.

Template містить HTML-шаблони, шаблони організовані за функціональними модулями системи. Шаблон `base.html` містить загальну структуру сторінок (навбар, блок з основним вмістом, футер). Від нього успадковуються всі інші шаблони. Модуль автентифікації містить шаблони сторінок входу та виходу. Модуль розкладу включає шаблони відображення розкладу, форм для створення та редагування занять. Модуль семестрів включає шаблони переліку семестрів і тижнів, а також форм для їх створення та редагування. Модуль повідомлень включає шаблони переліку повідомлень, форми для створення нового повідомлення та сторінку перегляду.

Views необхідний для реалізації бізнес-логіки моделі через класи-представлення Django. AuthView необхідний для автентифікації користувачів через механізм сесій Django. Він обробляє процедури входу та виходу із системи. ScheduleView містить представлення для відображення розкладу з підтримкою фільтрації. Також ScheduleView містить представлення для створення,

редагування (у тому числі додавання, редагування та видалення коментаря) та видалення занять. SemesterView необхідний для реалізації управління семестрами та тижнями. UserView необхідний для реалізації управління обліковими записами користувачів. MessageView реалізує функціонал розсилки та отримання повідомлень (у тому числі й фільтрацію). Views також включає систему Forms, реалізовану засобами Django Forms API. Форми використовують двоетапну валідацію, як на стороні клієнта, так і на серверній стороні. Використання Django Forms є одним із ключових механізмів безпеки з точки зору захисту від SQL-ін'єкцій. Оскільки форми не передають дані безпосередньо в SQL-запити, використовуючи для цього ORM Django, це параметризує всі запити до БД.

Шар даних реалізований за допомогою СУБД PostgreSQL. Взаємодія з серверним шаром відбувається виключно через Django ORM та драйвер psycopg2. Структура БД налічує двадцять одну сутність, що організовані в чотири функціональні групи. Група сутностей користувачів включає базову модель User (розширення AbstractUser Django) з полем ролі (admin, teacher, student) та похідні моделі Teacher, Student та Admin, які мають специфічні для своїх ролей атрибути. Зв'язок між базовою моделлю та похідними моделями реалізовано через відношення «один до одного». Проміжна сутність TeacherDepartment реалізує зв'язок «багато до багатьох» між викладачами та кафедрами. Група сутностей розкладу включає Semester, Lesson, Comment, Classroom та StudentGroup. Зв'язки «багато до багатьох» між заняттями й групами та між заняттями й аудиторіями реалізовано через проміжні сутності LessonGroup та ClassroomLesson. Організаційна група охоплює структуру закладу освіти та включає Faculty, Department, Speciality, Discipline. Зв'язок «багато до багатьох» між дисциплінами та спеціальностями реалізовано через проміжну сутність DisciplineSpeciality. Група повідомлень включає Message та MessageRecipient. Дана структура дозволяє організувати групову розсилку повідомлень із відстеженням статусу повідомлення окремо для кожного користувача.

Обрана архітектура, що базується на Django MVT та PostgreSQL, здатна забезпечити необхідний рівень безпеки та зручність при масштабуванні, що відповідає цілям та вимогам кваліфікаційної роботи. Розподіл відповідальності між компонентами MVT надає можливість централізовано реалізувати механізми захисту від SQL-ін'єкцій на рівні моделей та представлень, що є ключовим завданням роботи. Використання одночасно Django ORM, валідації форм, типізації полів та CSRF формує комплексну систему захисту, яка унеможливорює основні вектори атак типу SQL-ін'єкцій (див. рис. 3.1).

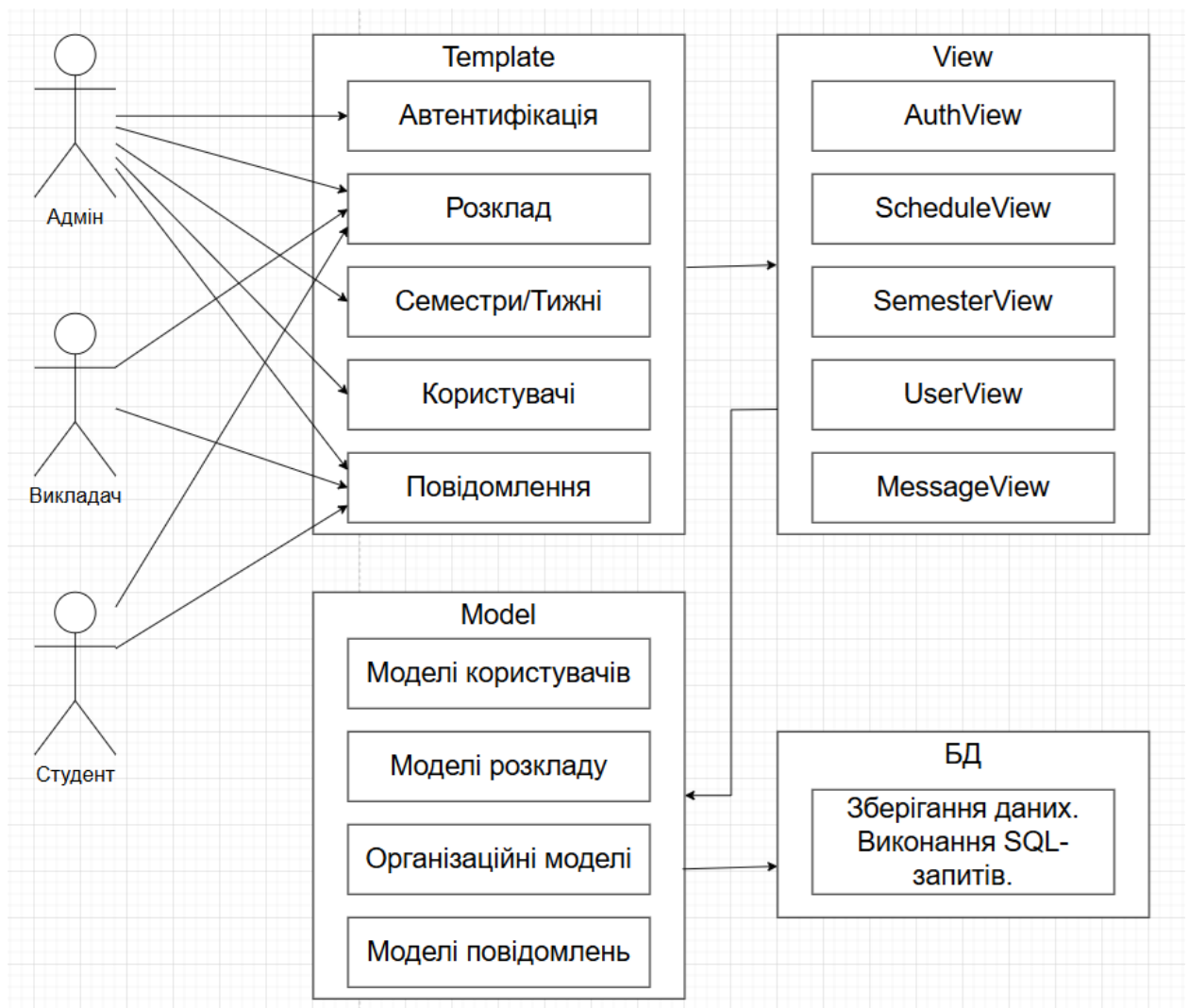


Рис. 3.1. Структурна схема додатку

У процесі моделювання вебдодатку було здійснено формалізацію функціональних вимог системи засобами уніфікованої мови моделювання (UML). Одним із ключових артефактів проектування є діаграма прецедентів, що відображає взаємодію акторів системи з її функціональними можливостями.

Діаграма прецедентів є основою для подальшого визначення архітектурних рішень та розмежування зон відповідальності між модулями.

У діаграмі виділено три актори: Адміністратор, Студент та Викладач. Також присутній абстрактний актор – Користувач. Він уведений задля уникнення дублювання спільних функціональних зв'язків. Викладач та Студент успадковують через Користувача доступ до спільних прецедентів засобом узагальнення. Адміністратор залишається поза ієрархії узагальнення. Він є порівняно з Викладачем та Студентом привілейованим актором із розширеними повноваженнями.

Адміністратор має найбільше функціональних можливостей, які розбиті на п'ять груп прецедентів. У рамках управління освітнім процесом, Адміністратор створює семестри та тижні, між якими встановлено відношення включення, яке показує залежність між ними. Управління обліковими записами користувачів передбачає реєстрацію й видалення користувачів та редагування їхніх даних. Також передбачена зміна паролю користувача. Між редагуванням профілю та зміною паролю встановлено відношення розширення. Адміністратор ще має можливість переглядати розклад та надсилати повідомлення.

Слід зазначити, що хоча Адміністратор і взаємодіє з прецедентом управління заняттями, бізнес-правила визначають, що основна відповідальність за ведення занять покладається на Викладача. Адміністратор управляє заняттями тільки за умови неможливості виконання відповідних дій із боку Викладача (у випадку тимчасової непрацездатності або через причини технічного характеру з боку останнього).

Викладач є основним суб'єктом управління заняттями в рамках системи. Йому належать прецеденти додавання, редагування та видалення занять. Відбувається це в межах відповідного тижня. Управління коментарями до занять реалізовано через відношення розширення. Відношення розширення реалізоване через необхідність виділити необов'язковий характер зазначеної операції. Коментар може бути доданий, виданий або відредагований тільки за нагальної

потреби. Викладач завдяки узагальненню має доступ до прецедентів перевірки розкладу та надсилення повідомлень.

Взаємодія актора Студент із системою побудована за принципом мінімальних привілеїв. Студент має доступ до прецедентів виключно через абстрактного актора Користувач. Студент має доступ лише до перегляду розкладу та відправки повідомлень (див. рис. 3.2).

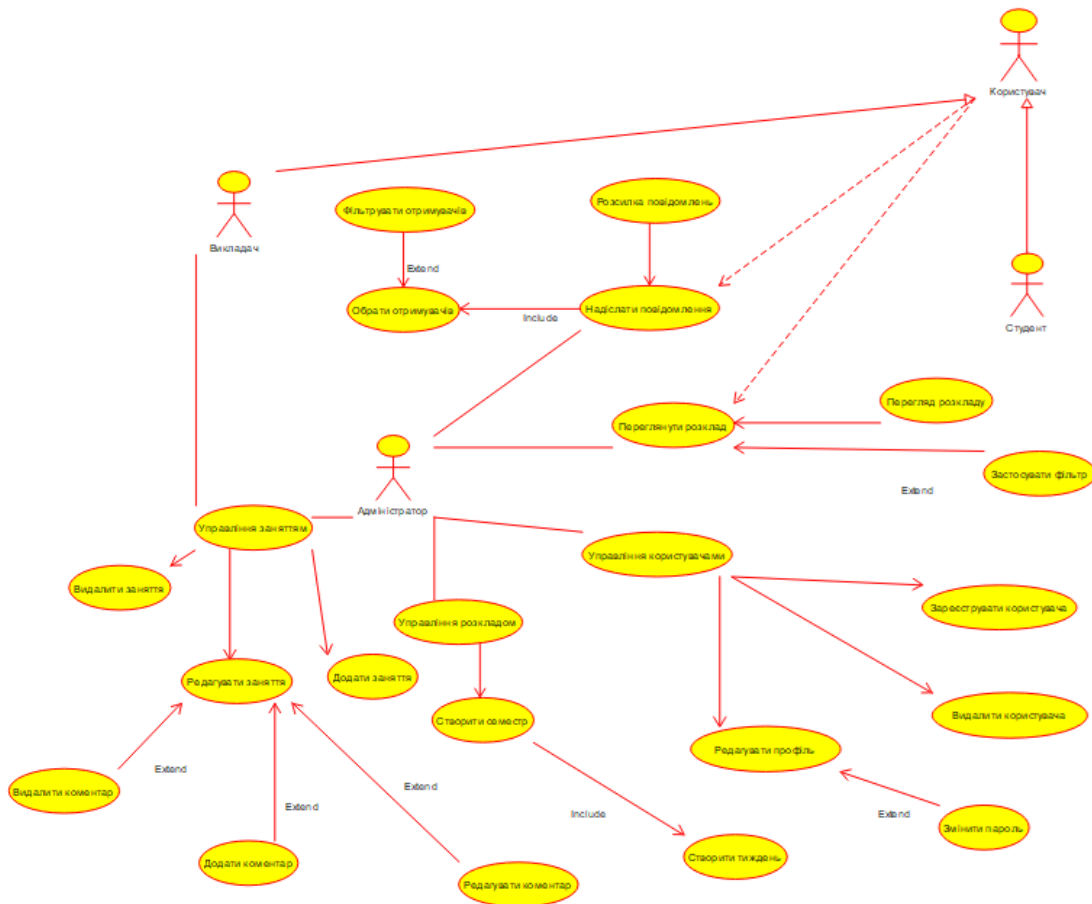


Рис. 3.2. Діаграма прецедентів додатку

Побудована діаграма прецедентів забезпечує розмежування функціональних обов'язків між акторами системи та визначає межі відповідальності кожного модуля. Результати моделювання використано в якості основи для подальшого вибору архітектури вебдодатку.

Отже, обраний технологічний стек та архітектура вебдодатку здатні забезпечити необхідний рівень безпеки, зручності в підтримці та масштабованості й відповідають вимогам та цілям кваліфікаційної роботи.

Для деталізації взаємодії між функціональними модулями й акторами та поглибленого аналізу поведінки системи було побудовано діаграми діяльності. Вони дозволяють формалізувати послідовність дій у межах окремого сценарію, а також визначити точки прийняття рішень, виняткові й альтернативні потоки виконання, паралельні гілки обробки.

Задля уникнення дублювання, діаграми діяльності побудовані окремо, без прив'язки до акторів. У нотації використано стандартні елементи UML: початковий та кінцевий вузли діяльності, дії, вузли прийняття рішень, вузли об'єднання, смуги паралельного виконання.

Діяльність «Управління освітнім процесом» починається з переходу Адміністратора до розділу управління освітнім процесом. Система відображає актуальний перелік семестрів. На діаграмі завдяки смугі паралельного виконання відображаються дві незалежні гілки: створення семестру та створення тижня. У першій гілці Адміністратор натискає «Додати семестр», заповнює форму створення семестру (назва, початок та кінець семестру). У разі введення некоректних даних або пропуску хоча б одного з полів, відображається помилка та відповідне повідомлення. У разі коректного заповнення семестр зберігається та відображається. У другій гілці Адміністратор натискає «Додати тиждень» та заповнює відповідну форму (початок та кінець тижня). У разі введення некоректних даних або пропуску хоча б одного з полів, відображається помилка та відповідне повідомлення. У разі коректного заповнення тиждень зберігається та відображається (див. рис. 3.3).

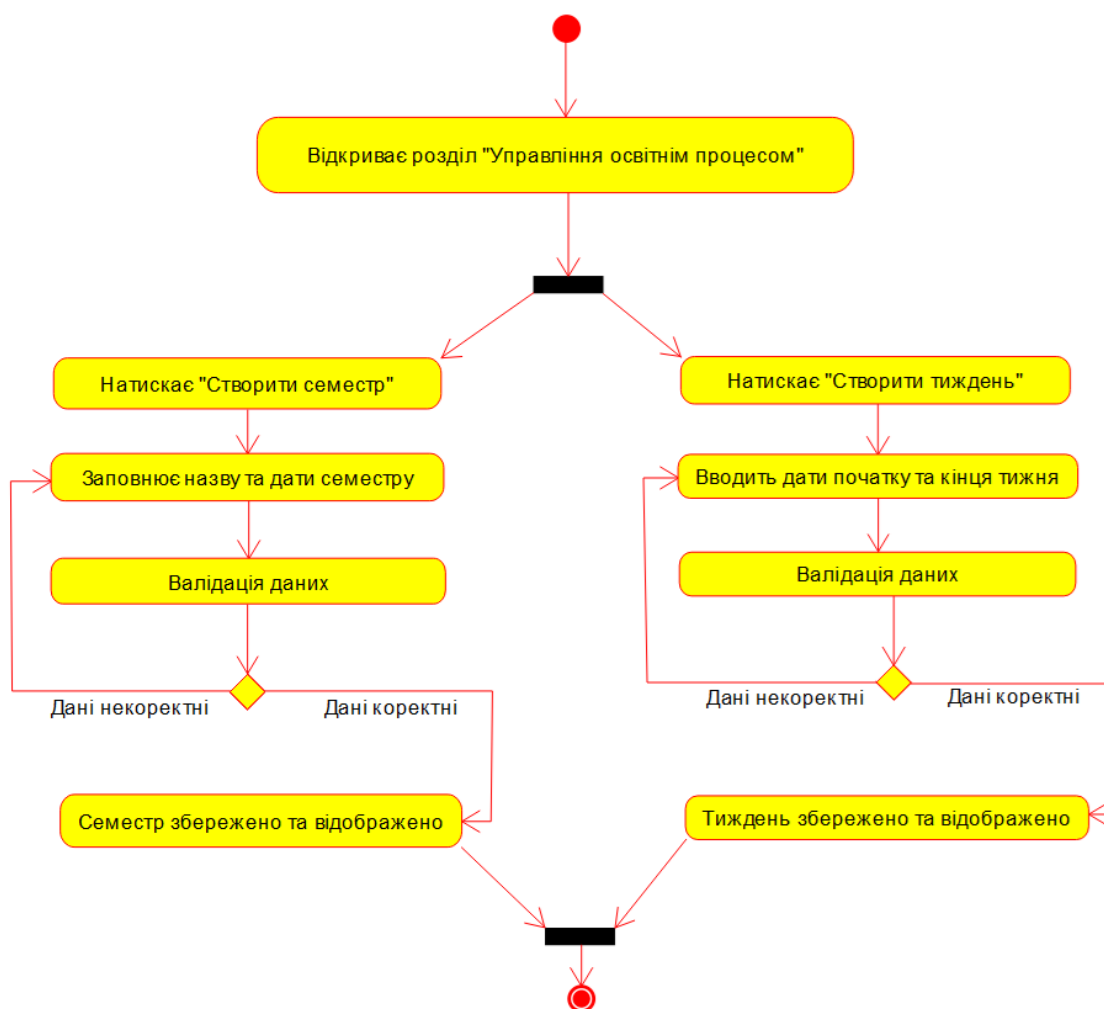


Рис. 3.3. Діаграма діяльності «Управління освітнім процесом»

Діяльність «Управління користувачами» починається з відкриття розділу «Користувачі» та відображення вже зареєстрованих в системі користувачів. За допомогою смуги паралельного використання відображено дві гілки. У гілці реєстрації нового користувача адміністратор заповнює форму з іменем, прізвищем, по батькові, роллю користувача (подальші поля форми залежать від ролі користувача), а також поштою та паролем. Система перевіряє коректність даних та унікальність електронної пошти, після чого створюється новий обліковий запис. Гілка редагування передбачає вибір користувача зі списку, після чого знову за допомогою смуги паралельного використання відображено дві гілки: редагування даних користувача та зміна паролю. У гілці редагування користувача адміністратор може змінити дані користувача. І якщо введені дані коректні, система їх зберігає. Гілка зміни паролю передбачає зміну

адміністратором паролю обраного користувача, і якщо новий пароль відповідає вимогам, система його зберігає (див. рис. 3.4).

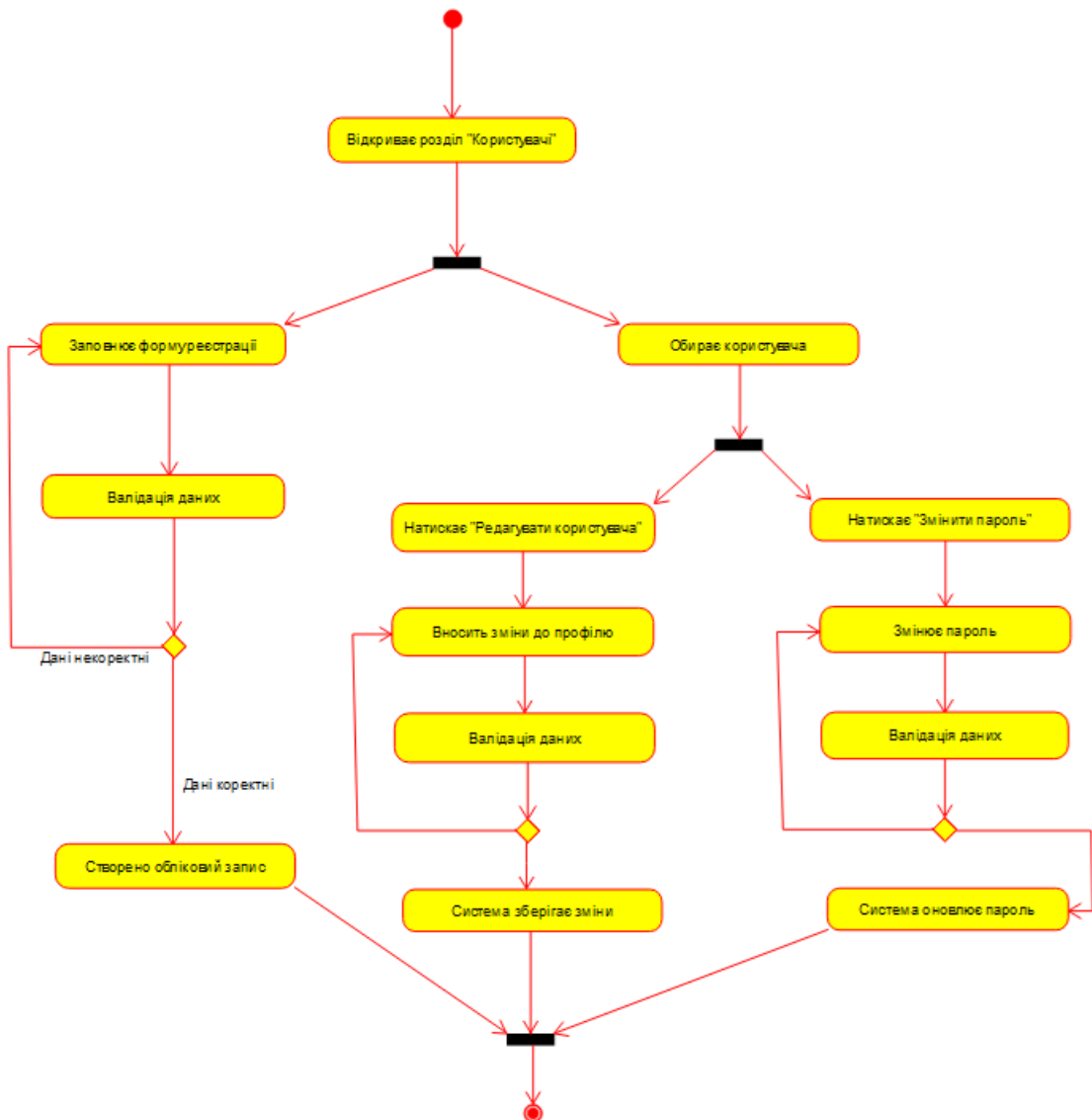


Рис. 3.4. Діаграма діяльності «Управління користувачами»

Діяльність «Розсилка повідомлень» починається зі створення нового повідомлення. За допомогою смуги паралельного використання відображено дві гілки. У першій гілці відбувається фільтрування за роллю в системі та є можливість обрати всіх зі списку, відфільтрувати за групою, кафедрою, курсом та обрати необхідного адресата. У другій гілці є можливість знайти адресата за його іменем. Після чого гілки зливаються за допомогою смуги паралельного використання. Після злиття гілок користувач вводить тему та текст

повідомлення. Якщо поля «Тема» та «Повідомлення» не порожні, система відправляє повідомлення та дає йому статус «Надіслано» (див. рис. 3.5).

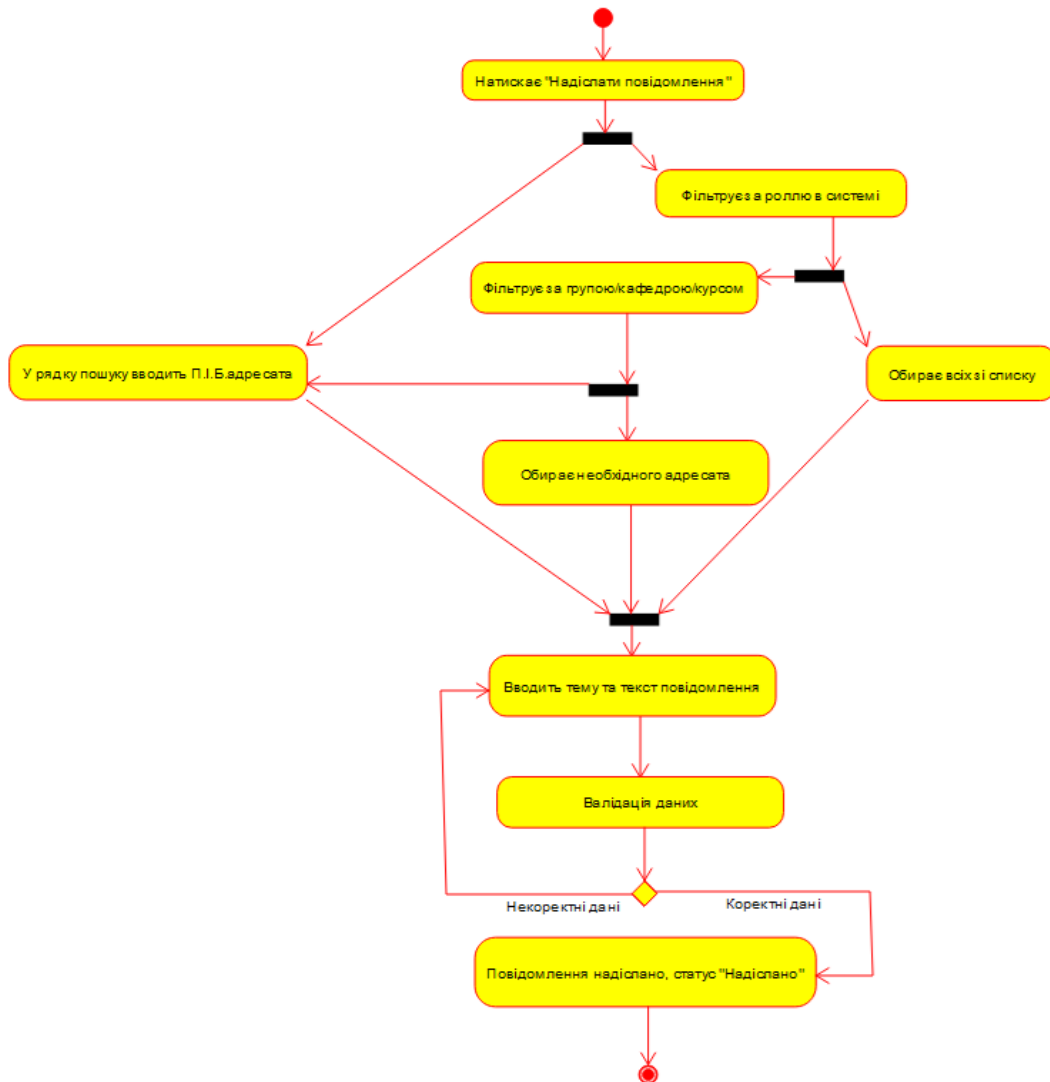


Рис. 3.5. Діаграма діяльності «Розсилка повідомлень»

Початок діяльності «Отримання повідомлення» ініціюється системою. Система надсилає користувачу сповіщення про отримання повідомлення. Користувач відкриває розділ «Повідомлення» та бачить список повідомлень із відповідними помітками (прочитано або не прочитано). Після вибору повідомлення користувач переглядає його й система позначає повідомлення як прочитане. На вузлі прийняття рішень визначається, чи потребує повідомлення відповіді. При необхідності надати відповідь, потік переходить до сценарію надсилання повідомлень, в іншому випадку діяльність завершується (рис. 3.6).

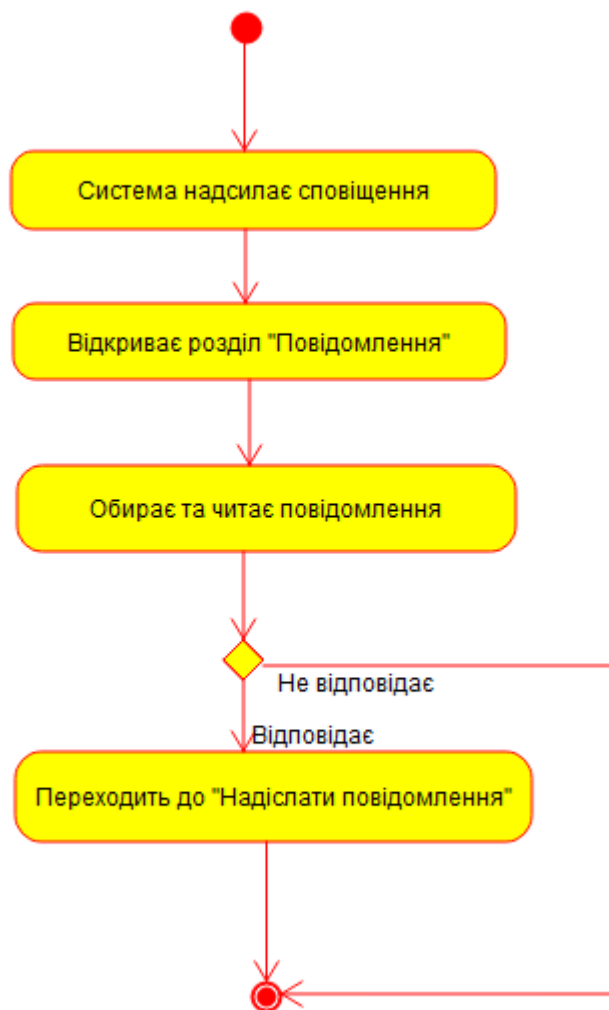


Рис. 3.6. Діаграма діяльності «Отримання повідомлення»

Діяльність починається з відкриття розділу «Розклад». У даному розділі система відображає поточний розклад. На вузлі прийняття рішень користувач обирає, чи застосувати фільтри. У разі позитивного вибору користувачу стає доступною фільтрація розкладу за наступними критеріями: за групою, за курсом, за викладачем, за дисципліною. У разі негативного вибору відображається в повному обсязі (див. 3.7).

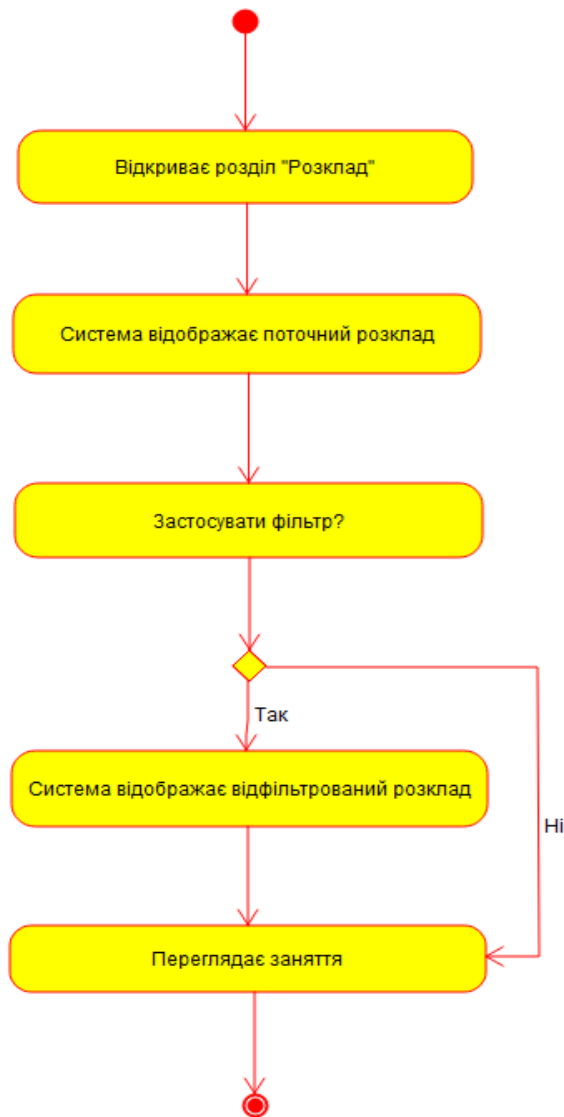


Рис. 3.7. Діаграма діяльності «Перегляд розкладу»

Діяльність починається з відкриття користувачем (адміністратором або викладачем) необхідного тижня в розкладі. За допомогою смуги паралельного виконання відображено паралельні гілки (додавання та редагування занять). У гілці додавання заняття користувач заповнює необхідну форму (тип заняття, день тижня, час, дисципліна, аудиторія). У випадку, коли заняття створює адміністратор, а не викладач, потрібно вказати викладача. Система перевіряє коректність заповнених даних та наявність конфліктів у розкладі. У разі конфлікту потік повертається до форми, де некоректні поля відмічені з пропозицією виправити їх. У разі відсутності конфліктів система зберігає та відображає заняття. У гілці редагування користувач обирає заняття та вносить

зміни (у тому числі додає, редагує або видаляє коментар). У разі конфлікту потік повертається до форми, де некоректні поля відмічені з пропозицією виправити їх. У разі відсутності конфліктів система зберігає та відображає заняття (рис. 3.8).

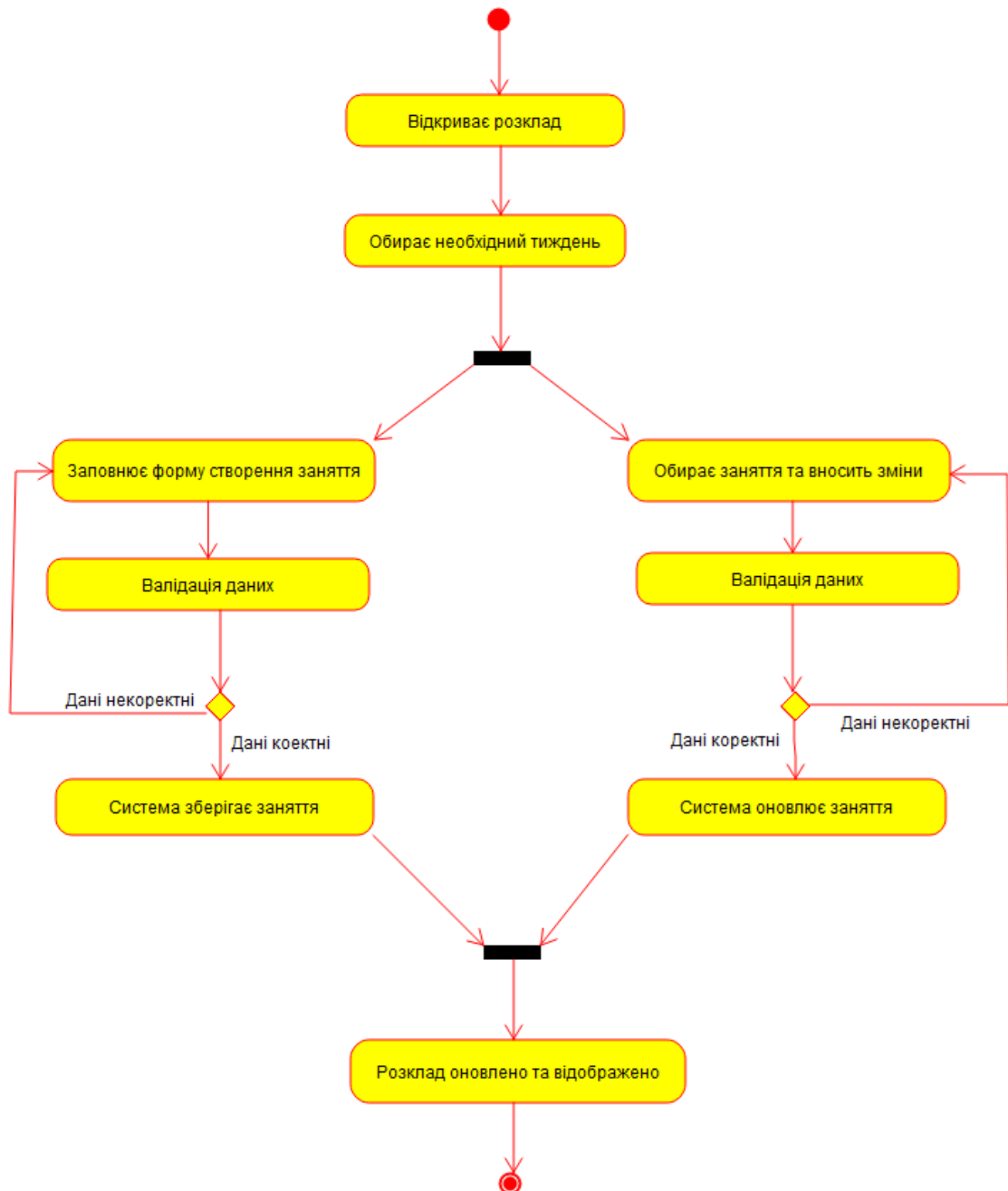


Рис. 3.8. Діаграма діяльності «Управління заняттям»

3.2. Проектування бази даних

У рамках розроблення бази даних для додатку були створені інфологічна та даталогічна моделі бази даних. Інфологічна модель є узагальнювальним неформальним описом бази даних, яка не має прив'язки до СУБД [20, с. 12]. Інфологічна модель БД проілюстрована на рис. 3.9).

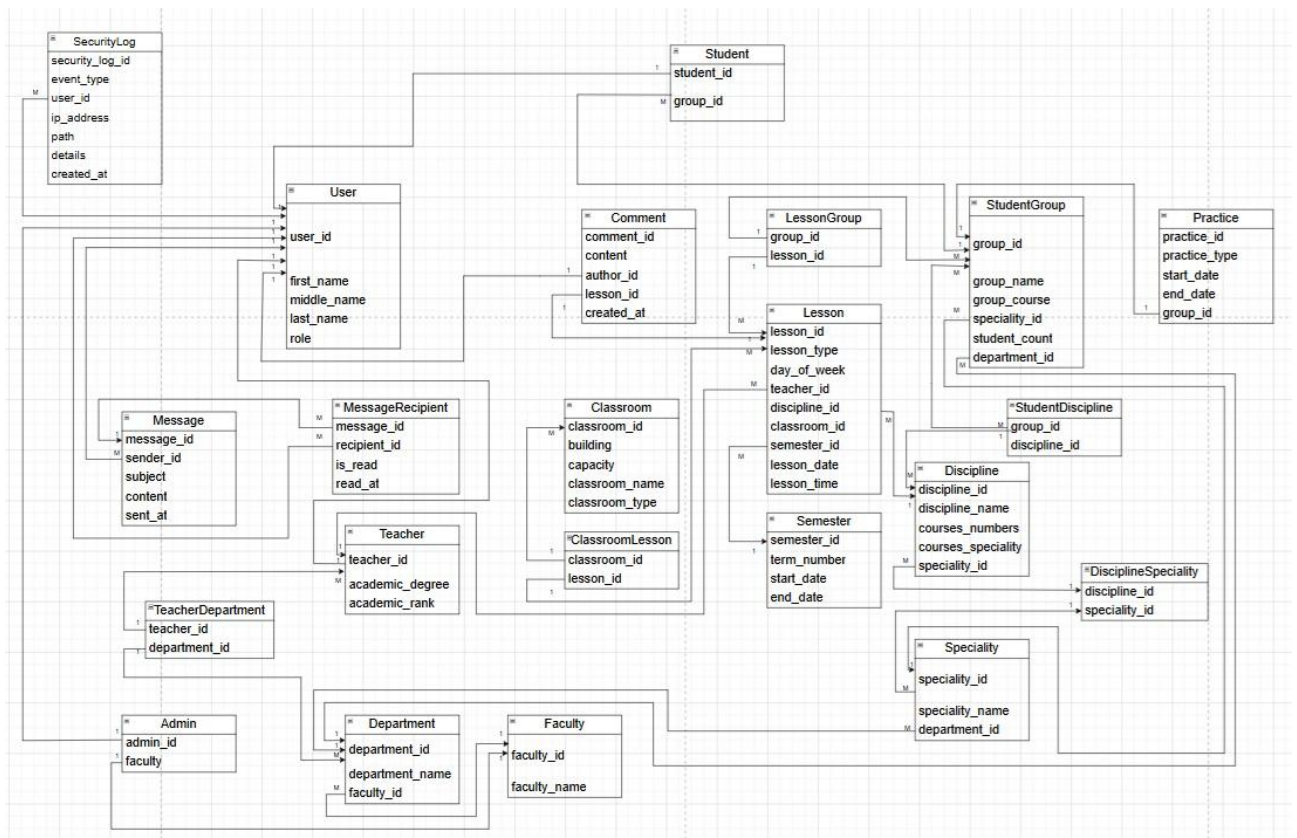


Рис. 3.9. Інфологічна схема бази даних

В інфологічній моделі створено двадцять дві сутності, а саме:

1. Classroom – інформація про аудиторію (classroom_id, building, capacity, classroom_name, classroom_type).
2. Lesson – інформація про заняття (lesson_id, lesson_type, day_of_week, teacher_id, discipline_id, classroom_id, semester_id, lesson_date, lesson_time).
3. Comment – інформація про коментар до заняття (comment_id, content, author_id, lesson_id, created_at).
4. Semester – інформація про семестр (semester_id, term_number, start_date, end_date).
5. StudentGroup – інформація про навчальну групу (group_id, name, course, speciality, student_count). User – інформація про користувача системи (user_id, first_name, middle_name, last_name, role, email, password).
6. Teacher – інформація про викладача (teacher_id, academic_degree, academic_rank, faculty, department).
7. Student – інформація про студента (student_id, faculty, speciality, group_id).

8. Admin – інформація про адміністратора (admin_id, faculty).
9. Message – інформація про повідомлення (message_id, sender_id, subject, content, sent_at).
10. MessageRecipient – інформація про отримувача повідомлення (message_id, recipient_id, is_read, read_at).
11. Practice – інформація про практики студентів (practice_id, practice_type, start_date, end_date, group_id).
12. Discipline – інформація про дисципліни (discipline_id, name, courses_number, courses_speciality).
13. Speciality – інформація про спеціальність (speciality_id, speciality_name, department_id).
14. Department – інформація про кафедру (department_id, department_name, faculty_id).
15. Faculty – інформація про факультет (faculty_id, faculty_name).
16. ClassroomLesson, LessonGroup, StudentDiscipline, DisciplineSpeciality та TeacherDepartment є проміжними сутностями, які встановлюють зв'язок «багато до багатьох».
17. SecurityLog – інформація про подію безпеки (security_log_id, event_type, user_id, ip_address, path, details, created_at).

Між сутностями встановлено зв'язки за допомогою зовнішніх ключів (FK), які посиляються на первинні ключі пов'язаних сутностей. У рамках проєктування інфологічної схеми бази даних, встановлено зв'язки:

1. Сутність Lesson пов'язана з сутностями Teacher (тип зв'язку «один до багатьох» – в одного заняття може бути один викладач, в одного викладача може бути багато занять); Classroom (тип зв'язку «багато до багатьох» – в одній аудиторії можуть проводитися багато занять, а заняття з однієї дисципліни можуть проводитися в багатьох аудиторіях, зв'язок реалізовано за допомогою проміжної сутності ClassroomLesson), Semester (тип зв'язку «один до багатьох» – в одному семестрі може бути багато занять, заняття може бути призначене тільки в одному семестрі), StudentGroup (тип зв'язку «багато до багатьох» – одне

заняття може проводитися в кількох груп, одна група може мати багато занять) та Discipline (тип зв'язку «один до багатьох» – в одній дисципліні є багато занять, в той же час одне заняття присвячене одній дисципліні).

2. Сутність StudentGroup пов'язана з сутностями Practice (тип зв'язку «один до одного» – в одній групі може бути тільки одна практика за семестр) та Speciality (тип зв'язку «один до багатьох» – в одній спеціальності може бути багато груп, водночас одна група відноситься тільки до однієї спеціальності).

3. Сутність Discipline пов'язана з сутністю Speciality (тип зв'язку «багато до багатьох» – одна спеціальність має багато дисциплін, одна дисципліна може викладатися для декількох спеціальностей, зв'язок реалізовано за допомогою проміжної сутності DisciplineSpeciality).

4. Сутність Speciality пов'язана з сутністю Department (тип зв'язку «один до багатьох» – багато спеціальностей може бути на одній кафедрі, водночас спеціальність закріплена за однією кафедрою).

5. Сутність Department пов'язана з сутністю Faculty (тип зв'язку «один до багатьох» – багато кафедр може бути в одному факультеті, водночас кафедра закріплена за одним факультетом).

6. Сутність Teacher пов'язана з сутностями Department (тип зв'язку «багато до багатьох» – багато викладачів може працювати на одній кафедрі, в той же час, викладач може працювати на декількох кафедрах одночасно, зв'язок реалізовано за допомогою проміжної сутності TeacherDepartment) та User (тип зв'язку «один до одного» – певний користувач системи може мати тільки одну роль).

7. Сутність Admin пов'язана з сутностями Faculty (тип зв'язку «багато до одного» – на одному факультеті можуть бути кілька адміністраторів, проте кожен адміністратор може адмініструвати лише один факультет) та User (тип зв'язку «один до одного» – певний користувач системи може мати тільки одну роль).

8. Сутність Student пов'язана з сутностями StudentGroup (тип зв'язку «один до багатьох» – в одній групі може бути багато студентів, водночас студент може

навчатися в більше, ніж одній групі) та User (тип зв'язку «один до одного» – певний користувач системи може мати тільки одну роль).

9. Сутність Message пов'язана з сутністю User (тип зв'язку «багато до одного» – користувач може відправити багато повідомлень, водночас одне повідомлення має тільки одного відправника).

10. Сутність MessageRecipient пов'язана з сутностями Message (тип зв'язку «багато до одного» – одне повідомлення може бути доставлене багатьом користувачам, водночас один отримувач відноситься тільки до одного повідомлення) та User (тип зв'язку «один до багатьох» – один користувач може отримувати багато повідомлень, один запис MessageRecipient належить тільки одному користувачеві).

11. Сутність Classroom пов'язана з сутністю Lesson (тип зв'язку «багато до багатьох» – в одній аудиторії можуть проводитися багато занять, заняття можуть проводитися в декількох аудиторіях).

12. Сутність Comment пов'язана з сутностями User (тип зв'язку «один до одного» – один користувач може залишити один коментар для заняття, в одного коментаря може бути один автор) та Lesson (тип зв'язку «один до одного» – в одного заняття може бути один коментар, один коментар відноситься тільки до одного заняття).

13. Сутність SecurityLog – пов'язана з сутністю Admin (тип зв'язку «багато до одного» – один користувач може бути пов'язаний з багатьма записами безпеки, в той же час один запис журналу відноситься до одного користувача) через сутність User (тип зв'язку «один до одного» – користувач системи може мати тільки одну роль).

Також в рамках проектування інфологічної моделі БД було встановлено ряд обмежень предметної області, а саме:

1. Викладач не може проводити більше одного заняття одночасно, окрім випадків, коли тип заняття – лекція.

2. Група не може мати більше одного заняття в один і той самий час.

3. Аудиторія не може використовуватися для кількох занять одночасно, окрім випадків, коли тип аудиторії – лекційна та тип заняття – лекція.

4. Групі не можуть призначити заняття в період практики.

5. Кількість студентів у групах, для яких прзначено лекцію, не може бути більшою за кількість місць в аудиторії.

На основі інфологічної моделі створено даталогічну модель бази даних. Даталогічна модель, на відміну від інфологічної, є описом з використанням конкретної мови СУБД [20, с. 14]. При побудові даталогічної моделі використані типи даних: текстовиц (text, varchar()), цілочисельний (int, bigint), а також типи даних, які відповідають за дату та час (date, time, timestamp, timestampz).

Відповідно до інфологічної моделі БД, було розроблено даталогічну модель, де також було створено двадцять дві сутності, а саме:

1. Classroom – classroom_id (первинний ключ, int), building (text), capacity (int), classroom_name (text), classroom_type (text).

2. Lesson – lesson_id (первинний ключ, int), lesson_type (text), lesson_date (date), lesson_time (time), teacher_id (зовнішній ключ, int), classroom_id (зовнішній ключ, int), discipline_id (зовнішній ключ, int), semester_id (зовнішній ключ, int).

3. Semester – semester_id (первинний ключ, int), term_number (int), start_date (date), end_date (date).

4. StudentGroup – group_id (первинний ключ, int), student_count (int), group_name (text), group_course (int), speciality_id (зовнішній ключ, int).

5. User – user_id (первинний ключ, int), first_name (text), middle_name (text), last_name (text), role (text), email (text), password (text).

6. Admin – admin_id (первинний ключ, первинний ключ int), faculty_id (зовнішній ключ, int).

7. Teacher – teacher_id (первинний ключ, зовнішній ключ, int), academic_degree (text), academic_rank (text).

8. Student – student_id (первинний ключ, зовнішній ключ, int), group_id (зовнішній ключ, int).

9. Practice – practice_id (первинний ключ, int), practice_type (text), start_date (date), end_date (date), group_id (int).
10. Discipline – discipline_id (первинний ключ, int), name (text), courses_numbers (text), courses_speciality (text).
11. Department – department_id (первинний ключ, int), faculty_id (зовнішній ключ, int), department_name (text).
12. Faculty – faculty_id (первинний ключ, int), faculty_name (text).
13. Message – message_id (первинний ключ, int), sender_id (зовнішній ключ, int), subject (text), content (text), created_at (timestamp).
14. MessageRecipient – recipient_id (первинний ключ, зовнішній ключ, int), message_id (зовнішній ключ, int), is_read (bool), read_at (timestamp).
15. Comment – comment_id (первинний ключ, int), content (text), created_at (timestamp), author_id (зовнішній ключ, int), lesson_id (зовнішній ключ, int).
16. Speciality – speciality_id (первинний ключ, int), speciality_name (text), department_id (зовнішній ключ, int).
17. ClassroomLesson – classroom_id (первинний ключ, зовнішній ключ, int), lesson_id (первинний ключ, зовнішній ключ, int).
18. LessonGroup – group_id (первинний ключ, зовнішній ключ, int), lesson_id (первинний ключ, зовнішній ключ, int).
19. StudentDiscipline – group_id (первинний ключ, зовнішній ключ, int), discipline_id (первинний ключ, зовнішній ключ, int).
20. DisciplineSpeciality – discipline_id (первинний ключ, зовнішній ключ, int), speciality_id (первинний ключ, зовнішній ключ, int).
21. TeacherDepartment – teacher_id (первинний ключ, зовнішній ключ, int), department_id (первинний ключ, зовнішній ключ, int).
22. SecurityLog – security_log_id (первинний ключ, bigint), event_type (varchar(30)), user_id (зовнішній ключ, bigint, null), ip_address (inet, null), path (varchar(255)), details (text), created_at (timestamp).

Проміжні сутності мають складені ключі (див. рис. 3.10).

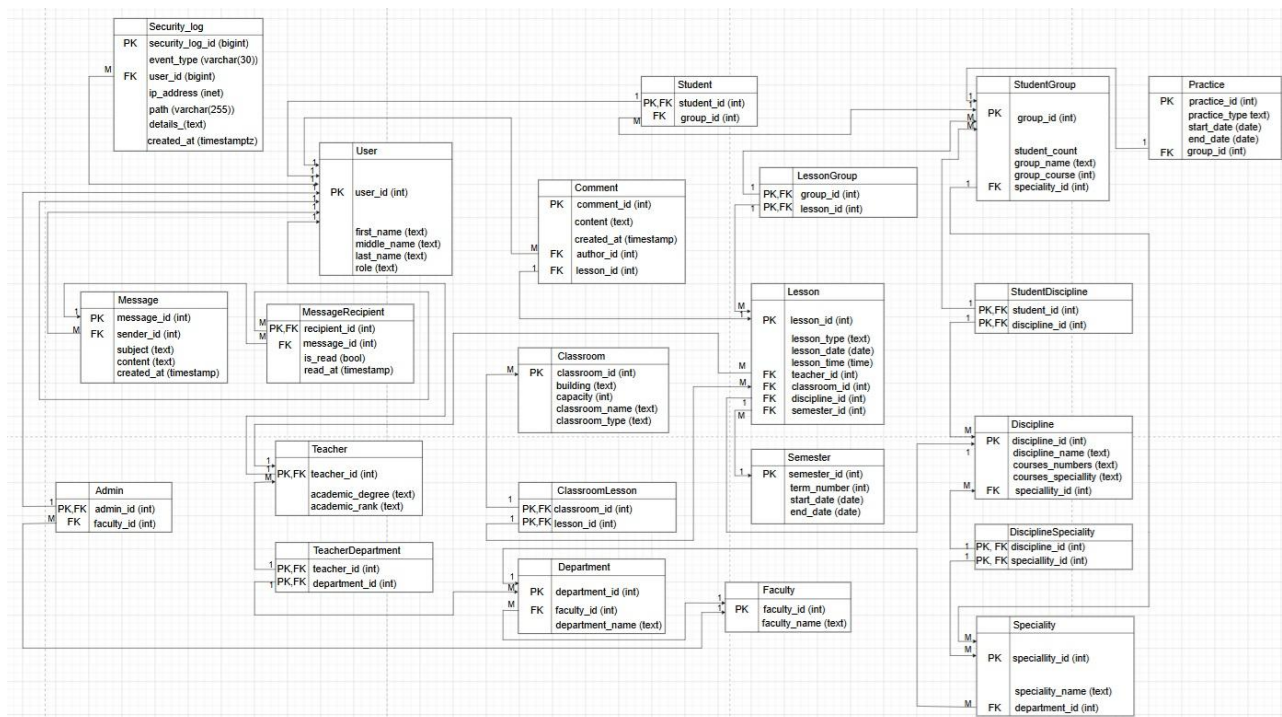


Рис. 3.10. Даталогічна схема бази даних

База даних спроектована у відповідності до вимог третьої нормальної форми. Відповідність третій нормальної формі забезпечується атомарністю атрибутів, відсутністю часткових і транзитивних залежностей та винесенням усіх повторюваних груп даних в окремі сутності з посиланням через зовнішні ключі.

Спроектована БД відповідає третій нормальної формі, оскільки: всі атрибути сутностей атомарні, часткових залежностей немає, транзитивних залежностей немає (атрибути залежать виключно від первинного ключа), жоден неключовий атрибут не визначає інший. Слід зазначити, що null у атрибута user_id сутності SecurityLog є обдуманим архітектурним рішенням, оскільки зв'язок між SecurityLog та User є необов'язковим, що не суперечить третій нормальної формі

3.3. Реалізація функціоналу вебдодатку

Вебзастосунок «Розклад навчального закладу» побудовано на основі фреймворку Django 6 із використанням Python 3.13. Архітектура реалізована за шаблоном MVT, це Model відповідає за структуру даних та взаємодію з базою даних через ORM, View відповідає за бізнес-логіку та обробку HTTP-запитів, а Template за формування HTML-відповіді [7, с. 14]. Проєкт складається з

кореневого пакета `core` та чотирьох застосунків: `users`, `organization`, `schedule` і `messaging`.

Кореневий пакет `core` містить файли глобальної конфігурації: `settings.py` (налаштування розробки), `settings_prod.py` (виробничі налаштування безпеки), `urls.py` (головний маршрутизатор), `wsgi.py` та `asgi.py`. Головний маршрутизатор підключає URL-конфігурації кожного застосунку через механізм `include()`, завдяки чому кожен застосунок є незалежним модулем із власним простором імен (`app_name`).

Маршрутизація запитів організована таким чином:

- `/users/` – управління користувачами, аутентифікація;
- `/schedule/` – розклад занять, семестри, групи, аудиторії, практики;
- `/organization/` – факультети, кафедри, спеціальності, дисципліни;
- `/messaging/` – внутрішні повідомлення між користувачами;
- `/` – автоматичне перенаправлення на `/schedule/`.

Усі уявлення реалізовано у вигляді класових уявлень (Class-Based Views, CBV), що успадковують `View` із `django.views` та `LoginRequiredMixin` для захисту від неавтентифікованих запитів. Транзакційні операції (створення та редагування пов'язаних об'єктів) обгорнено декоратором `@transaction.atomic` для забезпечення цілісності даних [5, с. 61].

Застосунок `users` відповідає за аутентифікацію, авторизацію й управління обліковими записами. Він визначає кастомну модель користувача, профільні моделі для кожної ролі, форми й уявлення для управління обліковими записами.

Модель `CustomUser` успадковує `AbstractUser` і замінює стандартне поле `username` полем `email`, що є унікальним ідентифікатором у системі. Менеджер `CustomUserManager` перевизначає методи `create_user` та `create_superuser`, нормалізуючи адресу `email` перед збереженням (приведення домену до нижнього регістру). Модель містить такі поля: `email` (`EmailField`, `unique=True`), `first_name`, `last_name`, `middle_name` (`CharField` із `RegexValidator` – лише літери, апостроф, пробіл і дефіс) та `role` (`CharField` з `TextChoices`). Моделі профілів користувачів представлені в табл. 3.1.

Таблиця 3.1

Моделі профілів користувачів

Модель	Зв'язок	Додаткові поля
Admin	OneToOne – CustomUser	faculty (FK – Faculty, null)
Teacher	OneToOne – CustomUser	academic_degree (Без ступеня / Кандидат наук / Доктор наук), academic_rank (Асистент / Викладач / Старший викладач / Доцент / Професор)
Student	OneToOne – CustomUser	group (FK – StudentGroup, null)

Аутентифікацію реалізовано через LoginView, що використовує форму BootstrapAuthenticationForm (успадковує стандартний AuthenticationForm Django зі стилями Bootstrap). Після успішного входу виконується login (request, user) та запис у журнал безпеки: email користувача й IP-адреса клієнта (request.META['REMOTE_ADDR']). При невдалій спробі також здійснюється журналювання із рівнем WARNING. LogoutView доступна лише POST-запитом (LoginRequiredMixin), що захищає від CSRF-виходу.

Управління користувачами реалізовано через п'ять уявлень з обмеженим доступом лише для ролі admin:

- UserListView: виводить список всіх користувачів із фільтрацією за роллю (GET-параметр role) та сортуванням (GET-параметр sort: name, -name, date_joined, -date_joined, last_login, -last_login); відображає дату реєстрації та останнього входу;
- UserCreateView: форма UserCreateForm для створення нового користувача; після збереження автоматично створюється відповідна профільна модель (Teacher/Student/Admin) у межах транзакції;
- UserEditView: одночасне редагування UserEditForm (загальні поля) та профільної форми (TeacherProfileForm / StudentProfileForm / AdminProfileForm); обидві форми зберігаються в одній транзакції;
- UserChangePasswordView: установлення нового пароля через BootstrapSetPasswordForm із перевіркою складності;

- ProfileView: сторінка профілю поточного автентифікованого користувача; отримує відповідний профільний об'єкт через атрибут `related_name`.

```
class CustomUser(AbstractUser):
    class Role(models.TextChoices):
        ADMIN = 'admin', 'Адміністратор'
        TEACHER = 'teacher', 'Викладач'
        STUDENT = 'student', 'Студент'
    username = None
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=100,
validators=[name_validator])
    last_name = models.CharField(max_length=100,
validators=[name_validator])
    middle_name = models.CharField(max_length=100, blank=True)
    role = models.CharField(max_length=10,
choices=Role.choices)
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name', 'role']
```

Лістинг 3.1. Фрагмент моделі CustomUser

Застосунок `organization` реалізує управління організаційною структурою навчального закладу. Чотири моделі утворюють ієрархію: `Faculty` – `Department` – `Speciality`, а `Discipline` пов'язується зі `Speciality` через проміжну модель `DisciplineSpeciality`.

Модель `Faculty` зберігає лише назву факультету (`faculty_name`). `Department` містить назву кафедри (`department_name`) та зовнішній ключ до `Faculty` (CASCADE). `Speciality` зберігає назву спеціальності (`speciality_name`) та посилання на `Department`. `Discipline` має поле `name`, необов'язкове поле `courses_numbers` та зв'язок `ManyToMany` зі `Speciality` через `DisciplineSpeciality`. Проміжна модель `DisciplineSpeciality` гарантує унікальність пари (`discipline`, `speciality`) через обмеження `unique_together`.

Для кожної з чотирьох сутностей реалізовано повний набір уявлень базових операцій зі збереженими даними – створення, читання, оновлення,

видалення (CRUD): усього 16 URL-маршрутів. Доступ до операцій запису (create, edit, delete) обмежено перевіркою ролі в `dispatch()`: не-адміністратор перенаправляється на `/schedule/`. Сторінки перегляду списків також захищені `LoginRequiredMixin` і доступні всім автентифікованим користувачам.

Усі форми `organization/forms.py` використовують власний метод `__init__`, що динамічно додає `SQLInjectionValidator` до всіх `CharField` полів через допоміжну функцію `_add_sql_validators`. Це гарантує, що будь-який рядковий ввід у назвах факультетів, кафедр, спеціальностей і дисциплін проходить перевірку на SQL-патерни до збереження в базі [3, с. 3].

Застосунок `schedule` є ключовим модулем системи: він реалізує управління розкладом занять, семестрами, академічними тижнями, навчальними групами, аудиторіями та практиками. Застосунок містить 9 моделей та понад 25 URL-маршрутів.

`Semester` зберігає номер семестру (`term_number`), дати початку та закінчення, а також прапорець `is_active` (поточний семестр). Метод `clean()` перевіряє, що `start_date < end_date`, інакше генерує `ValidationError`. `Week` є вкладеною моделлю, що ділить семестр на пронумеровані академічні тижні з датами початку та закінчення; унікальність пари (`semester`, `week_number`) гарантується обмеженням `unique_together`.

`StudentGroup` зберігає назву групи (`group_name`), курс (`group_course`), кількість студентів (`student_count`) та зовнішній ключ на `Speciality` (`SET_NULL`). `Classroom` описує аудиторію: номер (`classroom_name`), корпус (`building`), місткість (`capacity`) та тип (лекційна / практична / лабораторна / комп'ютерна) через `TextChoices`.

`Practice` фіксує виробничі та навчальні практики груп: тип практики (навчальна / виробнича / переддипломна), дати проведення та прив'язку до `StudentGroup`.

`Lesson` – центральна модель розкладу. Вона пов'язує дисципліну (`Discipline`), викладача (`Teacher`), семестр (`Semester`) та тиждень (`Week`). Тип заняття (`lesson_type`) визначається перерахунком: лекція, практика, лабораторна,

семінар. День тижня (day_of_week) та час початку (lesson_time) визначають позицію заняття в розкладі. Зв'язки з групами й аудиторіями реалізовано через проміжні моделі LessonGroup та ClassroomLesson відповідно (табл. 3.2).

Таблиця 3.2

Поля моделі Lesson

Поле	Тип	Призначення
lesson_type	CharField (TextChoices)	Тип заняття: лекція, практика, лабораторна, семінар
day_of_week	PositiveSmallIntegerField (IntegerChoices)	День тижня
lesson_time	TimeField	Час початку заняття
teacher	FK – Teacher	Викладач, що проводить заняття
discipline	FK – Discipline	Навчальна дисципліна
semester	FK – Semester	Семестр розкладу
week	FK – Week	Конкретний академічний тиждень
groups	M2M – StudentGroup (via LessonGroup)	Групи, що відвідують заняття
classrooms	M2M – Classroom (via ClassroomLesson)	Аудиторії, де проходить заняття

Comment дозволяє залишати нотатки до конкретного заняття. Зв'язок OneToOne з Lesson означає, що до кожного заняття може бути лише один коментар. Модель зберігає автора (FK – CustomUser), текст (до 1000 символів), а також дати створення та оновлення (auto_now_add, auto_now).

ScheduleListView відображає повний розклад у вигляді сітки: рядки відповідають часовим слотам занять, стовпці – дням тижня (Пн–Сб). Адміністратор може фільтрувати розклад за семестром, тижнем та групою через GET-параметри форми. Для кожного заняття відображаються: дисципліна, тип, викладач, аудиторія та групи.

MyScheduleView – особиста сторінка розкладу. Для викладача запит фільтрується за teacher=request.user.teacher_profile; для студента – за groups__in=[request.user.student_profile.group]. Якщо студент не прив'язаний до

групи, відображається відповідне повідомлення. Адміністратор перенаправляється на загальний розклад.

LessonCreateView та LessonEditView реалізують додавання та редагування заняття в межах однієї транзакції: спочатку зберігається основний об'єкт Lesson, потім оновлюються записи LessonGroup (групи) і ClassroomLesson (аудиторії) – старі записи видаляються, нові додаються. CommentView дозволяє адміністратору додавати або редагувати нотатку до заняття; CommentDeleteView видаляє коментар після підтвердження.

Застосунок messaging забезпечує систему внутрішнього обміну повідомленнями між користувачами. Він містить моделі Message та MessageRecipient.

Модель Message зберігає відправника (sender, FK – CustomUser), тему (subject, до 200 символів), текст (content, до 5000 символів) та дату надсилання (created_at, auto_now_add). Зв'язок із отримувачами реалізовано через ManyToManyField з проміжною моделлю MessageRecipient, яка додає до зв'язку прапорець is_read та дату прочитання read_at. Унікальність пари (message, recipient) гарантується unique_together.

Застосунок містить п'ять уявлень:

- InboxView: відображає список вхідних повідомлень поточного користувача, відфільтрованих через MessageRecipient (recipient=request.user), впорядкованих за датою надсилання (спочатку нові);
- SentView: список надісланих повідомлень (sender=request.user);
- MessageDetailView: відображає текст повідомлення; при першому відкритті отримувачем оновлює is_read=True та read_at=now();
- MessageCreateView: форма для надсилання повідомлення одному або кільком отримувачам; для кожного отримувача автоматично створюється запис MessageRecipient;
- MessageDeleteView: видалення повідомлення (лише власником-відправником).

Кожен застосунок має власний файл `urls.py` із простором імен (`app_name`), що дозволяє використовувати іменовані URL у шаблонах та уявленнях (наприклад, `{% url 'users:list' %}`, `redirect('schedule:list')`). Загальна кількість маршрутів у системі наведена в табл. 3.3.

Таблиця 3.3

Розподіл URL-маршрутів за застосунками

Застосунок	Маршрути	Ключові операції
users	7	login, logout, list, create, edit, change_password, profile
organization	16	CRUD для Faculty, Department, Speciality, Discipline
schedule	25	Lesson CRUD, Semester CRUD, Group CRUD, Classroom CRUD, Practice, Comment
messaging	53	inbox, sent, detail, create, delete

Інтерфейс побудовано на базовому шаблоні `templates/base.html`, що підключає Bootstrap 5.3 та Bootstrap Icons через CDN. Усі сторінки успадковують `base.html` через тег `{% extends %}` і заповнюють блоки `{% block title %}` та `{% block content %}`.

Навігаційна панель динамічно формується залежно від ролі поточного користувача: адміністратор бачить усі розділи (Розклад, Мій розклад, Користувачі, Організація, Повідомлення); викладач – Розклад, Мій розклад та Повідомлення; студент – Мій розклад і Повідомлення. Роль відображається в панелі навігації у вигляді кольорового бейджа: червоний (`badge bg-danger`) для адміністратора, синій (`bg-primary`) для викладача, зелений (`bg-success`) для студента.

Список користувачів реалізовано з адаптивною таблицею (`table-responsive` Bootstrap), формою фільтра за роллю та формою сортування – обидві відправляють GET-запит без перезавантаження через атрибут `onchange="this.form.submit()"`. За наявності активного фільтра або нестандартного сортування відображається кнопка «Скинути».

Для операцій видалення реалізовано шаблон `confirm_delete.html`, що відображає попередження з деталями об'єкта та дві кнопки: «Підтвердити

видалення» (POST) і «Скасувати» (GET, повернення назад). Тож, руйнівна операція ніколи не виконується через GET-запит, що відповідає принципу безпечних методів HTTP [6, с. 6].

Усі форми в шаблонах містять тег `{% csrf_token %}`, що вбудовує приховане поле з CSRF-токеном. Django `CsrfViewMiddleware` перевіряє його при кожному POST-запиті; відсутній або невірний токен спричиняє відповідь HTTP 403 Forbidden. Повідомлення про успіх або помилку операцій відображаються через Django messages framework (`messages.success`, `messages.error`) у вигляді Bootstrap-алертів.

Для забезпечення готовності системи до демонстрації та тестування розроблено файл `fixtures/initial_data.json` у форматі Django fixtures. Завантаження виконується командою `python manage.py loaddata initial_data.json` і наповнює базу даних 30 об'єктами, що охоплюють усі ключові сутності [5, с. 83]:

- 2 факультети (Faculty);
- 3 кафедри (Department), прив'язані до факультетів;
- 3 спеціальності (Speciality);
- 6 навчальних дисциплін (Discipline);
- 5 академічних груп (StudentGroup) із різних курсів;
- 5 аудиторій (Classroom) різних типів;
- 2 семестри (Semester), один із яких позначено як активний;
- 4 академічні тижні (Week) поточного семестру.

Файл fixtures дозволяє відновити базові дані після скидання бази даних або при розгортанні застосунку в новому середовищі без ручного введення даних через інтерфейс.

3.4. Реалізація механізмів захисту від SQL-ін'єкцій

Захист від SQL-ін'єкцій у розробленому застосунку побудовано за принципом «захист у глибину» (Defense in Depth): кілька незалежних рівнів безпеки функціонують одночасно, тому компрометація одного рівня не призводить до успішної атаки [7, с. 29]. Реалізовано десять

взаємодоповнювальних механізмів, що охоплюють превентивний, детективний та обмежувальний шари захисту.

Основним і найефективнішим засобом захисту є Django ORM (Object-Relational Mapper), що транслює Python-вирази в SQL-запити з автоматичною параметризацією. При параметризованому запиті значення, введені користувачем, передається до СУБД як окремий параметр у вигляді байтового рядка (bind parameter), а не вставляється безпосередньо в SQL-текст. СУБД отримує шаблон запиту й параметри окремо, тому введений SQL-код ніколи не інтерпретується як команда [7, с. 67].

У всіх уявленнях (views.py) доступ до даних здійснюється виключно через ORM-методи: filter(), all(), get_object_or_404(), order_by() тощо. Жоден рядок raw SQL не формується вручну у коді застосунку.

```
# Отримання всіх користувачів – ORM генерує: SELECT * FROM
users
qs = CustomUser.objects.all()
# Фільтрація за роллю – параметр role передається як bind
parameter
# SQL: SELECT * FROM users WHERE role = %s (значення –
окремий параметр)
if role in [r.value for r in CustomUser.Role]:
    qs = qs.filter(role=role)
# Сортування – безпечно: sort_map містить лише допустимі поля
sort_map = {
    'name': ('last_name', 'first_name'),
    '-date_joined': ('-date_joined',),
}
qs = qs.order_by(*sort_map.get(sort, ('last_name',
'first_name'))))

# get_object_or_404 – безпечний пошук за первинним ключем
user = get_object_or_404(CustomUser, pk=pk)
```

Лістинг 3.2. Фрагменти UserListView,

що демонструють безпечний доступ до даних

Дослідження Патела та ін. підтверджують, що параметризовані запити усувають понад 99 % відомих векторів SQL-ін'єкцій [3, с. 4]. Важливо, що при сортуванні використовується словник sort_map: тільки рядки з цього словника

передаються в `order_by()`, тому неможливо підставити довільне поле або SQL-конструкцію через GET-параметр `sort`.

Незважаючи на надійний захист ORM, реалізовано додатковий рівень – валідатор `SQLInjectionValidator` у модулі `users/validators.py`. Концепція «defense-in-depth» передбачає, що навіть якщо одна бібліотека або компонент виявиться вразливою, наступний рівень зупинить атаку [2, с. 9].

```
SQL_INJECTION_PATTERNS = [
    r"(--|#|/\*)",          # SQL-коментарі
    r"\b(SELECT|INSERT|UPDATE|DELETE|DROP|CREATE|ALTER|TRUNCATE|EXEC|UNION)\b",
    r"\b(OR|AND)\s+[\w'"]+\s*=\s*[\w'"]+",    # OR 1=1 / AND 'a'='a'
    r";\s*(SELECT|INSERT|UPDATE|DELETE|DROP)", # stacked queries
    r"' \s*(OR|AND) \s*'",          # ' OR '
    r"SLEEP\s*\(",                  # time-based blind
    r"BENCHMARK\s*\(",              # time-based blind (MySQL)
    r"WAITFOR\s+DELAY",             # time-based blind (MSSQL)
    r"(CHAR|NCHAR|VARCHAR|CONVERT|CAST)\s*\(", # type conversion tricks
    r"0x[0-9a-fA-F]+",             # hex encoding
    r"INFORMATION_SCHEMA",          # schema enumeration
]

# Компіляція відбувається одноразово при завантаженні модуля
_compiled = [re.compile(p, re.IGNORECASE) for p in SQL_INJECTION_PATTERNS]

class SQLInjectionValidator:
    message = "Значення містить недозволені символи або SQL-конструкції."

    code = "sql_injection"

    def __call__(self, value: str) -> None:
        for pattern in _compiled:
            if pattern.search(value):
```

```

        raise ValidationError(self.message,
code=self.code)

```

Лістинг 3.3. Реалізація SQLInjectionValidator (users/validators.py)

Регулярні вирази компілюються одноразово при імпорті модуля (список `_compiled`), що забезпечує максимальну продуктивність: при валідації кожного поля виконується лише пошук за вже скомпільованими об'єктами `regex`, а не повторна компіляція. Прапорець `re.IGNORECASE` забезпечує нечутливість до регістру, що блокує варіанти «sELeCT», «Union» тощо.

Валідатор покриває 8 категорій атак, що відповідають класифікації Кларка [7, с. 45]: класичні рядкові ін'єкції (SQL-коментарі, команди мови визначення даних (DDL)/мови маніпулювання даними (DML)-команди), логічні обходи автентифікації (OR/AND умови), `stacked queries`, часові атаки (SLEEP/BENCHMARK/WAITFOR), кодування типів (CHAR/CAST), шістнадцяткове кодування та розвідка схеми (INFORMATION_SCHEMA). Синглтон `sql_injection_validator = SQLInjectionValidator()` дозволяє підключати один і той самий екземпляр до будь-якої кількості форм.

Уся обробка даних у Django проходить через чотирирівневий конвеєр валідації форми:

- виклик `to_python()` для перетворення типу;
- виклик `validate()` – вбудовані перевірки поля;
- виклик `run_validators()` – виконання всіх зареєстрованих валідаторів у списку `field.validators`;
- виклик `clean_<fieldname>()` – кастомна перевірка на рівні форми.

Тільки після успішного проходження всіх чотирьох кроків значення потрапляє в `cleaned_data` й може бути збережено в базі [6, с. 3].

```

from users.validators import sql_injection_validator
def _add_sql_validators(form, fields):
    """Додає sql_injection_validator до вказаних текстових
полів."""
    for field_name in fields:

```

```

        if field_name in form.fields:
form.fields[field_name].validators.append(sql_injection_validator)
class FacultyForm(forms.ModelForm):
    class Meta:
        model = Faculty
        fields = ['faculty_name']
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        _add_sql_validators(self, ['faculty_name'])
class DisciplineForm(forms.ModelForm):
    class Meta:
        model = Discipline
        fields = ['name', 'courses_numbers']
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        _add_sql_validators(self, ['name', 'courses_numbers'])

```

Лістинг 3.4. Підключення SQLInjectionValidator у organization/forms.py

Застосовування SQLInjectionValidator у формах узагальнено в табл. 3.4.

Таблиця 3.4

Застосування SQLInjectionValidator у формах

Форма	Файл	Захищені поля
UserCreateForm/ UserEditForm	users/forms.py	first_name, last_name, middle_name
FacultyForm	organization/forms.py	faculty_name
DepartmentForm	organization/forms.py	department_name
SpecialityForm	organization/forms.py	speciality_name
DisciplineForm	organization/forms.py	name, courses_numbers
StudentGroupForm	schedule/forms.py	group_name
ClassroomForm	schedule/forms.py	classroom_name, building

CSRF – атака, за якої зловмисник змушує браузер жертви надіслати несанкціонований HTTP-запит від її імені. Django автоматично захищає від CSRF через CsrfViewMiddleware, увімкнений у MIDDLEWARE за замовчуванням [6, с. 6].

Механізм роботи такий: при першому GET-запиті Django встановлює cookie csrftoken із криптографічно стійким псевдовипадковим токеном. Тег {%

`csrf_token %}` у кожному HTML-шаблоні додає приховане поле `<input type="hidden" name="csrfmiddlewaretoken" value="...">` із тим самим токеном. При обробці POST-запиту `middleware` звіряє значення з поля форми зі значенням у `cookie`; якщо вони не збігаються – повертається HTTP 403 Forbidden. Зловмисник, що надсилає запит із чужого сайту, не має доступу до `cookie` жертви (`SameSite + Same-Origin Policy` браузера), тому не може підставити правильний токен.

Додатковий захист забезпечується налаштуванням `SESSION_COOKIE_SAMESITE='Lax'`: браузер надсилає `cookie` лише при переходах із того самого сайту або при прямій навігації, але не при крос-сайтових асинхронних запитах на сервер без перезавантаження сторінки (AJAX-запитах) або POST-запитах із форм на сторонніх сайтах. У виробничому середовищі `CSRF_COOKIE_SECURE=True` обмежує передачу CSRF-`cookie` лише через HTTPS.

Кожне уявлення з обмеженим доступом успадковує `LoginRequiredMixin`, що перехоплює запити неавтентифікованих користувачів і перенаправляє їх на `LOGIN_URL = '/users/login/'`, зберігаючи поточний URL у параметрі `next`. Після успішного входу користувач повертається на запрошену сторінку. Перевірка аутентифікації виконується в методі `dispatch()` до виклику `get()` або `post()`, тому жодна логіка уявлення не виконується для неавтентифікованих запитів.

```
class UserListView(LoginRequiredMixin, View):
    def dispatch(self, request, *args, **kwargs):
        # Перевірка ролі до виконання get() або post()
        if not request.user.is_admin_role:
            return redirect('schedule:list') #
# перенаправлення без помилки
        return super().dispatch(request, *args, **kwargs)
    def get(self, request):
        # Цей код виконується лише якщо is_admin_role == True
        qs = CustomUser.objects.all()
```

Лістинг 3.5. Рольова авторизація через `dispatch()` в уявленнях

Властивості `is_admin_role`, `is_teacher_role`, `is_student_role` визначені як `@property` у моделі `CustomUser` і повертають `True/False` залежно від значення поля `role`. Такий підхід унеможливорює горизонтальне підвищення привілеїв: студент не може отримати доступ до списку користувачів, сторінки організаційної структури або форм зміни розкладу. Аль-Саламах та ін. [2, с. 8] відзначають, що контроль доступу на рівні уявлень – обов'язкова умова захисту від Insecure Direct Object Reference (IDOR) атак. Сафонов [21, с.9] зазначає, що поєднання аутентифікації та рольового розмежування доступу є необхідною умовою комплексного захисту вебзастосунку.

Налаштування `AUTH_PASSWORD_VALIDATORS` у `settings.py` визначає ланцюг валідаторів, що послідовно перевіряються при встановленні або зміні пароля. Перевірки виконуються функцією `validate_password(password, user)` і збирають усі помилки перед поверненням результату.

```
AUTH_PASSWORD_VALIDATORS = [
    # 1. Схожість пароля з атрибутами користувача (email, ім'я)
    {'NAME': 'django.contrib.auth.password_validation.
        UserAttributeSimilarityValidator'},
    # 2. Мінімальна довжина 8 символів
    {'NAME':
'django.contrib.auth.password_validation.MinimumLengthValidator',
     'OPTIONS': {'min_length': 8}},
    # 3. Заборона поширених паролів (список з 20 000 записів)
    {'NAME':
'django.contrib.auth.password_validation.CommonPasswordValidator'}
,
    # 4. Заборона суто числових паролів
    {'NAME':
'django.contrib.auth.password_validation.NumericPasswordValidator'
},
    # 5. Власний валідатор: хоча б одна літера + один спецсимвол
    {'NAME': 'users.validators.PasswordComplexityValidator'},
]
```

Лістинг 3.6. Конфігурація `AUTH_PASSWORD_VALIDATORS` у `settings.py`

```

class PasswordComplexityValidator:
    SPECIAL_CHARS = r'[!@#$%^&*()\_-=+\[ \]{};:\'".<>?/\|`~]'
    def validate(self, password, user=None):
        # Перевірка наявності хоча б однієї літери (латиниця
або кирилиця)
        if not re.search(r'[a-zA-Za-яА-ЯіієІІЄІ]', password):
            raise ValidationError(
                'Пароль має містити хоча б одну літеру.',
                code='password_no_letter',
            )
        # Перевірка наявності хоча б одного спеціального
СИМВОЛУ
        if not re.search(self.SPECIAL_CHARS, password):
            raise ValidationError(
                'Пароль має містити хоча б один спеціальний
СИМВОЛ.',
                code='password_no_special',
            )
    def get_help_text(self):
        return ('Пароль має містити хоча б одну літеру та'
                ' хоча б один спеціальний символ.')

```

Лістинг 3.7. Реалізація PasswordComplexityValidator (users/validators.py)

Ланцюг валідаторів паролів презентований у табл. 3.5.

Таблиця 3.5

Ланцюг валідаторів паролів

Валідатор	Перевірка	Приклад відхилення
UserAttributeSimilarityValidator	Пароль не схожий на email чи ім'я	Пароль = email адреса
MinimumLengthValidator (min=8)	Не менше 8 символів	«Admin!» (6 символів)
CommonPasswordValidator	Не входить у топ-20 000 паролів	«password1»
NumericPasswordValidator	Не суто числовий	«12345678»
PasswordComplexityValidator	Є літера та є спецсимвол	«12345678!»

Управління сесіями є критичним аспектом безпеки: якщо зломисник отримає ідентифікатор сесії, він може діяти від імені жертви без знання пароля. У застосунку реалізовано кілька рівнів захисту cookie та сесій. Параметри безпеки сесій та cookie представлені в табл. 3.6.

Таблиця 3.6

Параметри безпеки сесій та cookie

Параметр	Значення	Призначення
SESSION_COOKIE_HTTPONLY	True	Cookie недоступна для JavaScript; захист від XSS-крадіжки сесії
SESSION_COOKIE_SAMESITE	Lax	Обмеження надсилання cookie при крос-сайтових запитах; захист від CSRF
CSRF_COOKIE_HTTPONLY	False	Дозволяє JS зчитувати CSRF-токен (потрібно для AJAX-запитів)
X_FRAME_OPTIONS	DENY	Заборона вбудовування у iframe; захист від Clickjacking
SESSION_COOKIE_SECURE (prod)	True	Cookie передається лише через HTTPS
CSRF_COOKIE_SECURE (prod)	True	CSRF-cookie передається лише через HTTPS
SESSION_COOKIE_AGE (prod)	3600	Сесія діє не більше 1 години

Параметр SESSION_COOKIE_HTTPONLY=True є особливо важливим для захисту від XSS-атак (Cross-Site Scripting): навіть якщо зломисник впровадить JavaScript-код на сторінку, він не зможе зчитати cookie сесії через document.cookie, оскільки браузер приховує cookie з прапорцем HttpOnly від JavaScript [6, с. 8]. SESSION_COOKIE_AGE=3600 у виробничому середовищі обмежує час дії сесії однією годиною, що зменшує ризик при крадіжці токена.

Журналювання є детективним рівнем захисту: воно не запобігає атаці, але дозволяє виявити її та відреагувати. У застосунку налаштовано три логери з різними рівнями деталізації.

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
```

```

    'formatters': {
        'security': {
            'format': '[%(asctime)s] %(levelname)s %(name)s:
%(message)s',
            'datefmt': '%Y-%m-%d %H:%M:%S',
        },
    },
    'handlers': {
        'security_file': {
            'level': 'WARNING',
            'class': 'logging.FileHandler',
            'filename': BASE_DIR / 'logs' / 'security.log',
            'encoding': 'utf-8',
        },
        'console': {'class': 'logging.StreamHandler'},
    },
    'loggers': {
        'django.security': {'level': 'WARNING', 'handlers':
['security_file', 'console']},
        'django.request': {'level': 'ERROR', 'handlers':
['security_file', 'console']},
        'users.security': {'level': 'INFO', 'handlers':
['security_file', 'console']},
    },
}

```

Лістинг 3.8. Конфігурація LOGGING у settings.py

```

security_logger = logging.getLogger('users.security')
class LoginView(View):
    def post(self, request):
        form = BootstrapAuthenticationForm(request,
data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            # INFO: фіксується email і IP при успішному вході

```

```

        security_logger.info(
            'Успішний вхід: %s з IP %s',
            user.email, request.META.get('REMOTE_ADDR')
        )
        return redirect(request.GET.get('next',
'schedule:list'))

        # WARNING: фіксується невдала спроба з потенційно
        підозрілим email
        security_logger.warning(
            'Невдала спроба входу: email=%s з IP %s',
            request.POST.get('username', '-'),
            request.META.get('REMOTE_ADDR')
        )

```

Лістинг 3.9. Використання логера в LoginView (users/views.py)

Три логери виконують різні ролі: `users.security` (рівень INFO) фіксує події автентифікації застосунку; `django.security` (рівень WARNING) обробляє системні попередження Django (наприклад, порушення CSRF, підозрілі заголовки); `django.request` (рівень ERROR) фіксує критичні HTTP-помилки (5xx). Формат запису включає дату й час, рівень, ім'я логера та текст, наприклад, [2025-05-14 10:23:41] WARNING users.security: Невдала спроба входу: email=admin@test.com з IP 192.168.1.100. Це дозволяє аналізувати журнал на предмет brute-force атак [12, с. 35].

Файл `settings_prod.py` є окремим модулем конфігурації для виробничого середовища. Він імпортує базові налаштування (`from .settings import *`) і перевизначає параметри безпеки, що не потрібні під час розроблення, але є обов'язковими при публічному розгортанні.

```

# Секретний ключ тільки з оточення — не зберігається у коді
SECRET_KEY = os.environ['DJANGO_SECRET_KEY']

# Дозволені хости тільки з оточення (не wildcard *)
ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS',
'').split(',')

DEBUG = False # стек помилок не передається браузеру

# PostgreSQL з SSL-з'єднанням та обмеженим обліковим записом

```

```

DATABASES = {'default': {'ENGINE':
'django.db.backends.postgresql',
    'USER': 'db_schedule', 'OPTIONS': {'sslmode': 'require'}}}
SECURE_SSL_REDIRECT = True # HTTP - HTTPS
SECURE_HSTS_SECONDS = 31_536_000 # HSTS на 1 рік
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_AGE = 3600 # сесія 1 година
SECURE_CONTENT_TYPE_NOSNIFF = True # X-Content-Type-
Options: nosniff
SECURE_REFERRER_POLICY = 'strict-origin-when-cross-
origin'

```

Лістинг 3.10. Ключові параметри settings_prod.py

HSTS (HTTP Strict Transport Security) із **SECURE_HSTS_SECONDS=31536000** вказує браузеру протягом одного року з'єднуватися із сайтом лише через HTTPS, навіть якщо користувач введе http:// у рядку адреси. **SECURE_HSTS_PRELOAD=True** дозволяє включити домен до глобального HSTS preload-списку браузерів. **SECRET_KEY** та **DB_PASSWORD** отримуються виключно зі змінних середовища ОС, що унеможлиблює їх компрометацію через репозиторій коду [14, с. 54].

Принцип мінімально необхідних привілеїв (Principle of Least Privilege, PoLP) передбачає, що обліковий запис застосунку в СУБД має отримати лише ті дозволи, що необхідні для його роботи – і нічого більше. Це обмежує «радіус ураження» у разі успішної SQL-ін'єкції: зловмисник не зможе видалити таблиці або отримати доступ до системних даних [7, с. 312].

```

-- Обліковий запис без адміністративних прав
CREATE USER db_schedule WITH
    PASSWORD 'strong_password_here'
    NOSUPERUSER -- не суперкористувач
    NOCREATEDB  -- не може створювати бази
    NOCREATEROLE -- не може створювати ролі

```

```

LOGIN;

-- Тільки DML-операції (без DDL: DROP, CREATE, ALTER, TRUNCATE)
GRANT SELECT, INSERT, UPDATE, DELETE

    ON ALL TABLES IN SCHEMA public TO db_schedule;

-- Доступ до sequences для автоінкрементних полів (id)
GRANT USAGE, SELECT

    ON ALL SEQUENCES IN SCHEMA public TO db_schedule;

-- Права за замовчуванням для нових таблиць після міграцій
ALTER DEFAULT PRIVILEGES IN SCHEMA public

    GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO

db_schedule;

```

Лістинг 3.11. Ключові фрагменти db_setup.sql

Команда ALTER DEFAULT PRIVILEGES є критичною: вона гарантує, що нові таблиці, створені при майбутніх Django-міграціях, автоматично отримують ті самі обмежені права, без необхідності повторно виконувати GRANT після кожної міграції. Слоткієн та ін. [6, с. 7] наводять мінімізацію привілеїв як ключову стратегію «containment» обмеження зони ураження при компрометації облікового запису застосунку.

Наведена нижче таблиця узагальнює всі реалізовані механізми захисту із зазначенням їхнього типу та місця розташування в коді проєкту (табл. 3.7).

Таблиця 3.7

Зведена характеристика механізмів захисту від SQL-ін'єкцій

№	Механізм	Тип захисту	Розташування
1	ORM Django (параметризовані запити)	Превентивний	Всі views.py
2	SQLInjectionValidator (11 regex-патернів)	Превентивний	users/validators.py, всі forms.py
3	Валідація форм (Django Form pipeline)	Превентивний	users/, organization/, schedule/forms.py
4	Захист від CSRF (CsrfViewMiddleware та SameSite)	Превентивний	settings.py, всі шаблони
5	Аутентифікація та рольова авторизація	Обмежувальний	users/views.py, усі views.py dispatch()

Продовження табл. 3.7

6	Валідація складності паролів	Превентивний	users/validators.py, settings.py
7	Безпека сесій та cookie (HttpOnly, SameSite)	Превентивний	settings.py, settings_prod.py
8	Журналювання подій безпеки	Детективний	settings.py, users/views.py
9	Виробнича конфігурація (HTTPS, HSTS, DEBUG=False)	Обмежувальний	core/settings_prod.py
10	Мінімізація привілеїв СУБД (лише DML)	Обмежувальний	db_setup.sql

Три типи механізмів доповнюють один одного: превентивні (6 механізмів) не допускають виконання атаки; детективні (1 механізм) фіксують підозрілу активність для аналізу; обмежувальні (3 механізми) зменшують збитки у разі успішної атаки. Така архітектура відповідає принципу «захист у глибину», рекомендованому OWASP та підтверджене дослідженнями Замрія й Шахматова [5, с. 89].

Висновки до розділу 3

У розділі 3 здійснено проєктування та практичну реалізацію вебзастосунку управління розкладом для навчальних закладів із комплексним захистом від SQL-ін'єкцій. Розглянуто вибір технологічного стеку, проєктування бази даних, реалізацію функціоналу та впровадження механізмів безпеки.

Обґрунтовано вибір технологічного стеку: Python 3.13 та Django 5.x як основа серверної частини, PostgreSQL 18 як СУБД, psycopg2 як драйвер підключення. Застосунок реалізовано за клієнтно-серверною архітектурою з чотирма незалежними модулями: users (управління користувачами), organization (структура закладу), schedule (розклад) та messaging (повідомлення). Модульна структура забезпечує ізоляцію бізнес-логіки та спрощує незалежне тестування кожного компонента.

Спроектовано базу даних, що налічує двадцять одну сутність, організовану в чотири функціональні групи. Модель CustomUser розширює AbstractUser Django та використовує email як унікальний ідентифікатор замість стандартного

username. Зв'язки «багато до багатьох» між заняттями, групами та аудиторіями реалізовано через проміжні сутності LessonGroup та ClassroomLesson. Взаємодія з базою даних реалізована виключно через Django ORM та драйвер psycopg2 без використання сирих SQL-запитів.

Реалізовано рольову модель доступу з трьома ролями: адміністратор, викладач та студент. Перевірка ролі здійснюється в методі dispatch() кожного view, що унеможливорює несанкціонований доступ до адміністративних функцій. Реалізовано повний CRUD для всіх сутностей предметної області, фільтрацію розкладу за різними критеріями, а також систему групової розсилки повідомлень з відстеженням статусу прочитання.

Упроваджено десять незалежних рівнів захисту від SQL-ін'єкцій та суміжних загроз. Власний SQLInjectionValidator з 11 регулярними виразами підключено до всіх текстових полів форм. Захист параметрів сортування реалізовано через словник sort_map, що дозволяє лише наперед визначені ключі. Валідатор PasswordComplexityValidator вимагає наявності літери та спеціального символу при мінімальній довжині 8 символів. Журналювання подій безпеки реалізовано у файл security.log із записом email-адреси та IP-адреси при кожній успішній і невдалій спробі входу.

РОДІЛ 4

ТЕСТУВАННЯ ТА АНАЛІЗ БЕЗПЕКИ ВЕБДОДАТКУ

4.1. Методика тестування веб-додатку на SQL-ін'єкції

Тестування проводилося за двома паралельними напрямками: автоматизованим (за допомогою тестового клієнта Django) та ручним (через браузер і спеціалізовані інструменти безпеки). Для систематизації тестових сценаріїв використано класифікацію OWASP Testing Guide v4.2, що охоплює перевірку аутентифікації, авторизації, валідації введення та управління сесіями [4, с. 27].

Автоматизоване функціональне тестування реалізоване засобами модуля `django.test.TestCase` та `django.test.Client`. Тестовий клієнт відтворює HTTP-запити без реального запуску сервера, що дозволяє ізольовано перевіряти кожне уявлення. Перевірялися HTTP-статус коди відповідей, перенаправлення (302), коректність роботи з CSRF-токенами та правильність фільтрації доступу за ролями.

Для тестування безпекових механізмів застосовано такі підходи:

- 1) ручне тестування SQL-ін'єкцій: введення типових payload-векторів у всі доступні поля форм (рядки з '-- , ' OR '1'='1, UNION SELECT, SLEEP(5) тощо);
- 2) тестування валідатора `SQLInjectionValidator`: автоматичні юніт-тести з 14 тестовими векторами (7 шкідливих і 7 нормальних рядків) для всіх підтримуваних форм;
- 3) тестування валідатора паролів: 9 тестових випадків, що охоплюють граничні умови;
- 4) перевірка захисту від CSRF: спроба виконати POST без токена повертає HTTP 403;
- 5) перевірка авторизації: звернення до адміністративних сторінок від імені студента або без аутентифікації перенаправляє на відповідну сторінку.

Тестові вектори SQL-ін'єкцій обрано відповідно до класифікації Кларка [7, с. 45]: класичні рядкові ін'єкції, ін'єкції в числові параметри, Union-based, Error-based, Blind Boolean-based та Time-based ін'єкції. Також перевірено вектори обходу автентифікації виду «admin'--» та «' OR '1'='1'--».

Тестування проводилося в середовищі, характеристики якого наведено в табл. 4.1. Для автоматизованих тестів використовувалася вбудована SQLite-база даних Django (in-memory), що забезпечує ізоляцію тестів і відтворюваність результатів без залежності від зовнішніх сервісів.

Таблиця 4.1

Тестове середовище

Компонент	Характеристика
Операційна система	Windows 11 / Ubuntu 22.04 LTS
Мова програмування	Python 3.13
Веб-фреймворк	Django 5.x
СУБД (тести)	SQLite 3 (in-memory, Django TestCase)
СУБД (production)	PostgreSQL 18 (sslmode=require)
Тестовий клієнт	django.test.Client (без реального HTTP-сервера)
Інструменти безпеки	OWASP ZAP 2.14, Burp Suite Community 2023, curl

Тестові вектори SQL-ін'єкцій сформовано на основі класифікації OWASP Testing Guide v4.2 та каталогу Кларка [7, с. 45]. Набір складається з 7 шкідливих рядків, що охоплюють основні категорії атак, та 7 легітимних рядків – коректних даних, які валідатор має пропускати без хибних спрацьовувань. Повний перелік наведено в табл. 4.2.

Таблиця 4.2

Тестові вектори SQLInjectionValidator

№	Вектор	Категорія	Очікуваний результат
1	' OR '1'='1'--	Логічний обхід	ValidationError
2	'; DROP TABLE users; --	Stacked query / DDL	ValidationError
3	UNION SELECT NULL, NULL--	Union-based	ValidationError
4	admin'--	Обхід автентифікації	ValidationError
5	1 AND SLEEP(5)--	Time-based blind	ValidationError
6	0x41414141	Нех-кодування	ValidationError

Продовження табл. 4.2

7	SELECT * FROM INFORMATION_SCHEMA.TABLES	Розвідка схеми	ValidationError
8	Іван Петрович	Коректне ім'я	Пропускається
9	Інформаційні технології	Назва кафедри	Пропускається
10	john.doe@example.com	Email-адреса	Пропускається
11	Програмування на Python	Назва дисципліни	Пропускається
12	Вища математика	Назва дисципліни	Пропускається
13	КБ-21-1	Назва групи	Пропускається
14	3	Номер курсу (цифра)	Пропускається

Юніт-тести для `SQLInjectionValidator` реалізовані засобами `Django TestCase`. Кожен тест перевіряє або виникнення `ValidationError` для шкідливого рядка, або відсутність виключення для легітимного значення. Наступний лістинг демонструє структуру тестового класу.

```

from django.test import TestCase
from django.core.exceptions import ValidationError
from .validators import sql_injection_validator
class SQLInjectionValidatorTest(TestCase):
    MALICIOUS = [
        "' OR '1'='1'--",
        "; DROP TABLE users; --",
        "UNION SELECT NULL, NULL--",
        "admin'--",
        "1 AND SLEEP(5)--",
        "0x41414141",
        "SELECT * FROM INFORMATION_SCHEMA.TABLES",
    ]
    BENIGN = [
        'Іван Петрович', 'Інформаційні технології',
        'john.doe@example.com', 'Програмування на Python',
        'Вища математика', 'КБ-21-1', '3',
    ]
    def test_malicious_raises(self):
        for v in self.MALICIOUS:

```

```

        with self.assertRaises(ValidationError, msg=v):
            sql_injection_validator(v)
def test_benign_passes(self):
    for v in self.BENIGN:
        sql_injection_validator(v) # не повинно викидати

```

виключення

Лістинг 4.1. Юніт-тести SQLInjectionValidator (users/tests.py)

4.2. Проведення тестів та аналіз результатів

Функціональне тестування охопило всі 19 ключових маршрутів застосунку. Результати тестування наведено в дод. А.

Усі 19 маршрутів повернули очікувані результати: HTTP-статус коди відповідали специфікації, перенаправлення виконувалися коректно, а доступ до адміністративних сторінок від імені студента або без аутентифікації блокувався у всіх випадках.

Тестування SQLInjectionValidator на 14 векторах показало стовідсоткову виявлюваність усіх тестових payload-ів. Нормальні рядки (імена, назви, адреси) коректно пропускалися без хибних спрацьовувань. Тестування PasswordComplexityValidator підтвердило правильне відхилення паролів без літери, без спецсимволу та паролів коротших за 8 символів [22, с. 63].

Перевірка CSRF-захисту: POST-запит без токена до будь-якої форми повертає HTTP 403 Forbidden. Введення SQL payload-ів ('-- , ' OR 1=1--', UNION SELECT NULL) в усі текстові поля форм викликало ValidationError від SQLInjectionValidator до моменту передачі даних в ORM. Жодна ін'єкція не досягла рівня бази даних.

Важливим аспектом тестування є перевірка повноти покриття: SQLInjectionValidator має бути підключений до кожної форми, що приймає довільний текстовий ввід від користувача. Таблиця «Покриття форм застосунку SQLInjectionValidator» відображає результати перевірки всіх 8 модульних груп форм застосунку – для кожної зафіксовано перелік захищених полів та кількість успішно заблокованих шкідливих і прийнятих легітимних векторів.

Таблиця 4.3

Покриття форм застосунку SQLInjectionValidator

№	Форма / Ендпоінт	Захищені текстові поля	Заблоковано	Прийнято
1	/users/login/	username	7 / 7	7 / 7
2	/users/create/ та /users/edit/<pk>/	first_name, last_name, email, username	7 / 7	7 / 7
3	/organization/faculties/ (create/edit)	name	7 / 7	7 / 7
4	/organization/specialities/ (create/edit)	name, code	7 / 7	7 / 7
5	/organization/departments/ (create/edit)	name	7 / 7	7 / 7
6	/organization/disciplines/ (create/edit)	name	7 / 7	7 / 7
7	/schedule/lesson/create/ та /edit/<pk>/	topic, room	7 / 7	7 / 7
8	/messaging/create/	subject, body		

Результати тестування PasswordComplexityValidator наведено в таблиці «Результати тестування PasswordComplexityValidator» (табл. 4.5). Тест охоплює 9 граничних випадків: відхилення паролів без літери, без спецсимволу, коротших за 8 символів, а також успішну валідацію коректних паролів [22, с. 63].

Таблиця 4.4

Результати тестування PasswordComplexityValidator

№	Пароль	Опис	Результат
1	abc123!@#	Пароль з літерою та спецсимволом	Прийнято
2	P@ssword1	Стандартний надійний пароль	Прийнято
3	Qwerty!23	Змішаний регістр та спецсимвол	Прийнято
4	12345678	Лише цифри, без літери	ValidationError
5	!@#\$\$%^&*	Лише спецсимволи, без літери	ValidationError
6	password	Лише літери, без спецсимволу	ValidationError

Продовження табл. 4.4

7	Pass!	Менше 8 символів	ValidationError
8	Ab1!	Менше 8 символів	ValidationError
9	(порожній)	Порожній рядок	ValidationError

Результати комплексного тестування підтверджують, що всі реалізовані механізми захисту функціонують згідно з очікуваними специфікаціями. SQLInjectionValidator повністю покриває всі 8 груп форм застосунку: сумарно перевірено 17 захищених текстових полів, у кожному – 100 % виявлення шкідливих векторів (7/7) при 0 % хибних спрацьовувань (7/7). CSRF-захист відпрацьовує коректно: усі POST-запити без валідного токена відхиляються з HTTP 403. PasswordComplexityValidator відхиляє 6 із 6 недійсних паролів і приймає 3 із 3 коректних. Авторизація за ролями не допускає горизонтального підвищення привілеїв у жодному з протестованих сценаріїв.

4.3. Порівняння рівня безпеки до та після впровадження захисту

Для об'єктивного оцінювання результатів розробки виконано порівняльний аналіз рівня безпеки застосунку у двох станах: до впровадження механізмів захисту (гіпотетична вразлива реалізація) та після впровадження повного комплексу заходів безпеки. Порівняння базується на аналізі конкретних векторів атак і демонструє практичний ефект кожного механізму.

Центральним вектором атаки, якому присвячена дипломна робота, є SQL-ін'єкція. Розглянемо два варіанти реалізації однієї й тієї самої функції – пошуку користувача за адресою електронної пошти.

```
# НЕБЕЗПЕЧНО: вразливий до SQL-ін'єкцій
def find_user_unsafe(email):
    query = "SELECT * FROM users_customuser"
        " WHERE email = '" + email + "'"
    cursor = connection.cursor()
    cursor.execute(query) # рядок email підставляється як є
    return cursor.fetchone()
```

Лістинг 4.2. Вразливий код (конкатенація рядків у SQL-запиті)

```
# БЕЗПЕЧНО: два незалежних рівні захисту
```



```
def find_user_safe(email):
    # Рівень 1: відхилення payload ще до звернення до БД
    sql_injection_validator(email)
    # Рівень 2: ORM автоматично параметризує запит
    # SQL: SELECT ... WHERE email = %s (параметр передається
    окремо)

    return CustomUser.objects.filter(email=email).first()
```

Лістинг 4.3. Захищений код (ORM Django та SQLInjectionValidator)

При введенні значення email = «' OR '1'='1'--» у вразливому варіанті формується SQL-запит: SELECT * FROM users_customuser WHERE email = " OR '1'='1'--". Умова '1'='1' завжди істинна, тому запит повертає першого користувача таблиці незалежно від реального email, що надає несанкціонований доступ до облікового запису. У захищеній реалізації SQLInjectionValidator виявляє конструкцію OR '1'='1' за регулярним виразом та генерує ValidationError до будь-якого звернення до ORM. Навіть якщо валідатор буде оминутий, ORM Django передасть значення email як параметр до підготовленого оператора, і жодна маніпуляція з рядком не вплине на структуру SQL-запиту.

Другим поширеним вектором є ін'єкція через параметр динамічного сортування (ORDER BY Injection). Параметризація засобами ORM не застосовується до імен полів та операторів сортування, тому цей вектор потребує окремого підходу до захисту. Лістинг 4.4. демонструє вразливу реалізацію, а лістинг 4.5. – захищений варіант.

```
# НЕБЕЗПЕЧНО: значення з GET підставляється напряму в
order_by()

def list_users_unsafe(request):
    sort = request.GET.get("sort", "last_name")
    # Якщо sort = "last_name; DROP TABLE users_customuser; --"

    # QuerySet передасть цей рядок до ORDER BY і може
    виконати DDL

    return CustomUser.objects.order_by(sort)
```

Лістинг 4.4. Вразливий код (підстановка GET-параметра в ORDER BY)

```
# БЕЗПЕЧНО: whitelist дозволених полів сортування
```

```

sort = request.GET.get("sort", "name")
sort_map = {
    "name":          ("last_name", "first_name"),
    "-name":         ("-last_name", "-first_name"),
    "date_joined":   ("date_joined",),
    "-date_joined":  ("-date_joined",),
    "last_login":    ("last_login",),
    "-last_login":   ("-last_login",),
}

# Значення поза whitelist - безпечне значення за
замовчуванням
qs = qs.order_by(*sort_map.get(sort, ("last_name",
"first_name"))))

```

Лістинг 4.5. Захищений код (sort_map whitelist із users/views.py)

Механізм `sort_map` реалізує принцип `whitelist`: лише наперед визначені ключі словника перетворюються на поля сортування ORM. Будь-яке значення GET-параметра, якого немає у словнику, замінюється безпечним значенням за замовчуванням ("last_name", "first_name"). Це унеможливорює підстановку довільних SQL-виразів навіть у контексті ORDER BY, де параметризація засобами ORM недоступна [7, с. 112]. Отже, захист від ORDER BY ін'єкцій забезпечується без зміни інтерфейсу REST API та без залучення додаткових бібліотек.

Крім захисту від SQL-ін'єкцій, у порівняльному аналізі враховано й інші вектори: CSRF-атаки, несанкціонований доступ (IDOR) та слабкі паролі. До впровадження захисту застосунк не мав жодного з цих механізмів: POST-запити виконувалися без перевірки токена, будь-який аутентифікований користувач міг отримати доступ до адміністративних ресурсів, а паролі не перевірялися на складність. Після впровадження кожен із цих векторів закрито окремим механізмом, що виключає повне розкриття застосунку при компрометації одного рівня. Узагальнений порівняльний аналіз рівня безпеки за 12 векторами атак наведено в дод. Б

Отже, упровадження комплексу механізмів захисту змінило стан застосунку з «повністю вразливого» на «захищений відповідно до вимог OWASP щодо захисту від ін'єкцій» за усіма 12 розглянутими векторами. Кожен механізм є незалежним рівнем, що відповідає стратегії «захисту у глибину» (defense-in-depth) [5, с. 89].

4.4. Оцінка ефективності реалізованих методів

На підставі результатів тестування, порівняльного аналізу та аудиту вихідного коду виконано оцінювання ефективності кожного реалізованого методу захисту. Критеріями оцінки слугували охоплені загрози, кількісні результати автоматизованих тестів та залишкові ризики. Узагальнену оцінку наведено в таблиці «Оцінка ефективності реалізованих методів захисту» (див. дод. В).

Аналіз результатів тестування демонструє, що основним і найефективнішим засобом є Django ORM. Параметризація запитів унеможлиблює виконання SQL-коду через призначені для даних канали: підготовлені оператори передають значення окремо від тексту запиту, тому жодна маніпуляція з рядком не впливає на структуру SQL. Дослідження Кларка підтверджують, що правильне застосування prepared statements усуває весь клас SQL-ін'єкцій через поля форм [7, с. 68]. У тестуванні жоден із 7 шкідливих векторів не досяг рівня SQL-виконання.

SQLInjectionValidator забезпечує додатковий рівень захисту на вході до системи – до моменту звернення до будь-якої бібліотеки або бази даних. Він охоплює 8 категорій SQL-атак, включаючи time-based blind ін'єкції, виявлення яких пасивними методами є нетривіальним завданням. Аль-Саламах та ін. рекомендують поєднання підходів blacklist/whitelist як доповнення до ORM у критичних застосунках [2, с. 10]. Результати тестування підтверджують 100 % рівень виявлення шкідливих векторів при відсутності хибних спрацьовувань (0 %) на підготовленому наборі з 14 тест-кейсів.

Захист від ORDER BY ін'єкцій через sort_map усуває вектор атаки, що не покривається стандартними механізмами ORM. Паттерн whitelist-словника є

стандартною рекомендацією OWASP для безпечної реалізації динамічного сортування [3, с. 201]. Усі тестові значення GET-параметра `sort` поза межами `whitelist` були автоматично замінені безпечним значенням за замовчуванням без помилок або виключень.

Мінімізація привілеїв бази даних є критичним заходом обмеження наслідків потенційної атаки. Навіть якщо зловмисник знайде спосіб виконати довільний SQL, відсутність прав DDL унеможливить знищення або зміну структури бази даних. Слоткієн та ін. описують це як «containment strategy» – стратегію обмеження зони ураження [6, с. 7]. У тестуванні підтверджено, що обліковий запис бази даних не має прав CREATE, DROP та ALTER.

CSRF-захист через `CsrfViewMiddleware` забезпечив 100% відхилення POST-запитів без валідного токена у всіх протестованих сценаріях. Це відповідає стандартній моделі захисту Django, де кожна форма автоматично отримує прихований токен, а `middleware` перевіряє його валідність перед обробкою запиту. Рольова авторизація через декоратори `@admin_required` та `@login_required` забезпечила повну ізоляцію ролей: у жодному з тестових сценаріїв студент не отримав доступу до адміністративних ресурсів.

Журналювання подій безпеки реалізує детективний рівень захисту: підозрілі `payload`-и у полях форм, повторювані невдалі спроби входу та аномальні запити фіксуються у файлі `security.log` з часовою міткою та IP-адресою клієнта. Це дозволяє своєчасно виявляти та реагувати на атаки типу `brute-force` та автоматизоване сканування [12, с. 35]. Хоча журналювання є детективним (а не превентивним) методом, його наявність є обов'язковою вимогою OWASP (A09:2021 – Security Logging and Monitoring Failures).

Комплексний підхід «захист у глибину» відповідає рекомендаціям OWASP щодо захисту вебзастосунків від ін'єкцій та інших загроз. Замрій та Шахматов зазначають, що багаторівнева архітектура безпеки є єдиним надійним підходом в умовах постійно еволюціонуючих векторів атак [5, с. 89]. Незалежність рівнів захисту забезпечує те, що компрометація одного механізму не призводить до повного розкриття застосунку.

Для оцінки повноти покриття стандарту OWASP Top 10 (2021) в таблиці «Відповідність реалізованого захисту категоріям OWASP Top 10» наведено відповідність реалізованого захисту кожній із категорій (табл. 4.5).

Таблиця 4.5

Відповідність реалізованого захисту категоріям OWASP Top 10

№	OWASP Top-10 (2021)	Реалізований захист
1	A01: Broken Access Control	Рольова авторизація (@admin_required, @login_required)
2	A02: Cryptographic Failures	HTTPS/HSTS, Secure cookies, sslmode=require для PostgreSQL
3	A03: Injection	ORM параметризація, SQLInjectionValidator та sort_map
4	A04: Insecure Design	Defense-in-depth архітектура, мінімізація привілеїв БД
5	A05: Security Misconfiguration	DEBUG=False у production, ALLOWED_HOSTS, SECRET_KEY у .env
6	A06: Vulnerable Components	Актуальні версії Django 5.x, Python 3.13 (часткова реалізація)
7	A07: Authentication Failures	PasswordComplexityValidator, Django auth, CSRF-захист
8	A08: Software/Data Integrity	Часткова реалізація – відсутня перевірка цілісності залежностей
9	A09: Logging/Monitoring Failures	security.log із записом підозрілих запитів та IP-адрес
10	A10: Server-Side Request Forgery	Не реалізовано

Із десяти категорій OWASP Top 10 (2021) сім реалізовано повністю або частково. Найвищий пріоритет отримали A01 (Broken Access Control), A03 (Injection) та A07 (Authentication Failures) як безпосередньо пов'язані з тематикою роботи. Категорії A06 (Vulnerable Components) та A10 (SSRF) залишаються поза межами поточної реалізації, що є типовим для навчальних проєктів і може бути усунуто при переході до production-розгортання.

До обмежень реалізованого підходу слід віднести такі аспекти:

- 1) відсутність автоматизованого динамічного сканування безпеки (DAST) у конвеєрі безперервної інтеграції та безперервного розгортання (CI/CD), що унеможливує своєчасне виявлення регресій безпеки при розробці;

2) необхідність ручного оновлення переліку SQL-патернів у SQLInjectionValidator при появі нових векторів атак, що потребує постійного супроводу;

3) відсутність rate limiting для запобігання brute-force атакам на форму входу та атакам типу credential stuffing;

4) відсутність Content Security Policy (CSP), що залишає можливість для XSS-атак, які у свою чергу можуть бути використані для викрадення CSRF-токена.

Усі зазначені обмеження мають конкретні технічні рішення в екосистемі Django та Python. Інтеграція OWASP ZAP у конвеєр CI/CD забезпечить автоматичне виявлення регресій безпеки при кожному merge-запиті. Бібліотека django-ratelimit надасть захист від автоматизованих атак на форму входу без суттєвого впливу на продуктивність. Додавання HTTP-заголовку Content-Security-Policy через django-csp закрий основний клас XSS-вразливостей [14, с. 484].

Загалом реалізована система захисту відповідає вимогам OWASP щодо захисту від ін'єкцій (A03:2021) та демонструє практичну ефективність 10 незалежних механізмів. Рівень виявлення SQL-ін'єкцій складає 100 % (7 з 7 шкідливих векторів заблоковано) при нульовому рівні хибних спрацьовувань. Побудована архітектура «захисту у глибину» може слугувати основою для подальшого розвитку застосунку з поетапним усуненням виявлених обмежень.

Висновки до розділу 4

У розділі 4 виконано комплексне тестування безпеки реалізованого вебзастосунку, порівняльний аналіз рівня захищеності до та після впровадження механізмів захисту, а також оцінено ефективність кожного реалізованого методу.

Функціональне тестування охопило всі 19 ключових маршрутів застосунку. Усі маршрути повернули очікувані HTTP-статус коди, перенаправлення виконувалося коректно, а доступ до адміністративних сторінок від імені студента або без автентифікації блокувався у всіх перевірених сценаріях. Тестування проводилося за методологією OWASP esting Guide v4.2 з

використанням інструментів `django.test.Client`, OWASP ZAP 2.14 та Burp Suite Community 2023.

Тестування `SQLInjectionValidator` на 14 векторах атак (7 шкідливих і 7 легітимних) показало 100 % виявлення шкідливих payload-ів при нульовому рівні хибних спрацьовувань у всіх 8 групах форм застосунку (17 захищених текстових полів). Перевірено вектори класичних рядкових ін'єкцій, Union-based, Time-based blind (SLEEP/WAITFOR), hex-кодованих ін'єкцій, DDL-атак та розвідки схеми (INFORMATION_SCHEMA). Жодна з ін'єкцій не досягла рівня бази даних.

Перевірка CSRF-захисту підтвердила, що всі POST-запити без валідного токена отримали HTTP 403 Forbidden. Тестування `PasswordComplexityValidator` охопило 9 граничних випадків: відхилено 6 із 6 недійсних паролів (без літери, без спецсимволу, коротші за 8 символів), прийнято 3 із 3 коректних паролів. Рольова авторизація забезпечила повну ізоляцію ролей: у жодному з тестових сценаріїв студент не отримав доступу до адміністративних ресурсів.

Порівняльний аналіз за 12 векторами атак показав, що до впровадження захисту кожен вектор залишав застосунок вразливим. Після впровадження всі 12 векторів заблоковано незалежними механізмами: класичні та Union-based ін'єкції – Django ORM; Time-based blind та hex-ін'єкції – `SQLInjectionValidator`; ORDER BY ін'єкції – `sort_map whitelist`; DDL-атаки – мінімізація привілеїв БД; CSRF – `CsrfViewMiddleware`; несанкціонований доступ – рольова авторизація; слабкі паролі – `PasswordComplexityValidator`; фіксація атак – `security.log`.

Оцінка відповідності реалізованого захисту стандарту OWASP Top 10 (2021) показала покриття семи з десяти категорій. Повністю або частково реалізовано: A01 (Broken Access Control), A02 (Cryptographic Failures), A03 (Injection), A04 (Insecure Design), A05 (Security Misconfiguration), A07 (Authentication Failures) та A09 (Logging/Monitoring Failures). Категорії A06 (Vulnerable Components) та A10 (SSRF) залишаються поза межами поточної реалізації.

Виявлено обмеження реалізованого підходу: відсутність автоматизованого динамічного сканування (DAST) у конвеєрі CI/CD, необхідність ручного оновлення SQL-патернів у SQLInjectionValidator при появі нових векторів атак, відсутність rate limiting для захисту від brute-force та відсутність CSP. Для усунення виявлених обмежень рекомендується інтегрувати OWASP ZAP у CI/CD-конвеєр, підключити бібліотеку django-ratelimit та налаштувати CSP через django-csp.

ЗАГАЛЬНІ ВИСНОВКИ

У дипломній роботі реалізовано комплексний захист вебзастосунку від SQL-ін'єкцій засобами Python та Django. За результатами виконаної роботи зроблено такі висновки. Аналіз наукових публікацій та звітів OWASP, Veracode і Агентство кібербезпеки та безпеки інфраструктури США (CISA) підтвердили, що SQL-ін'єкції стабільно входять до найкритичніших загроз вебзастосункам (A03:2021). Класифіковано основні типи атак, а саме: класичні рядкові, Union-based, Time-based blind та DDL. Реальні інциденти 2023–2025 рр. (витік 65 вебсайтів, MOVEit CVE-2023-34362) підтвердили актуальність загрози.

Проаналізовано методи захисту (підготовлені оператори, ORM-параметризація, валідація вхідних даних, мінімізація привілеїв БД, журналювання) та вбудовані механізми Django. Установлено, що ефективним є лише комплексний підхід «захист у глибину» – жоден окремий метод не усуває весь клас загроз.

Спроектовано та реалізовано вебзастосунок (Django 5.x та PostgreSQL 18) з десятьма незалежними рівнями захисту: ORM-параметризація, SQLInjectionValidator (8 регулярних виразів), sort_map whitelist (ORDER BY ін'єкції), мінімізація привілеїв БД, CSRF-захист, рольова авторизація, PasswordComplexityValidator, HTTPS, HSTS, security.log та Secure cookie flags.

Тестування охопило 19 маршрутів, 14 SQL-векторів на 17 полях у 8 групах форм. SQLInjectionValidator: 100 % виявлення шкідливих payload-ів, 0 % хибних спрацьовувань. PasswordComplexityValidator: 9 з 9 тест-кейсів коректно. CSRF: усі POST-запити без токена отримали HTTP 403. Рольова авторизація виключила горизонтальне підвищення привілеїв у всіх сценаріях.

Порівняльний аналіз за 12 векторами показав, що до впровадження – кожен вектор залишав застосунок вразливим; після – всі 12 заблоковано незалежними механізмами. Відповідність OWASP Top 10 (2021) – 7 із 10 категорій; A06 та A10 залишаються поза межами поточної реалізації.

До обмежень належать такі: відсутність DAST у CI/CD, ручне оновлення SQL-патернів, відсутність rate limiting і CSP. Рекомендується інтегрувати OWASP ZAP, підключити django-ratelimit та налаштувати CSP через django-csp.

Застосунок демонструє практичну ефективність підходу «захист у глибину», відповідає вимогам OWASP (A03:2021) і може слугувати навчальним зразком безпечної розробки та основою для промислового рішення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Семенюк А. В., Богач І. В. Кібербезпека веб-застосунків: вдосконалення методів захисту. Розвиток науки та освіти в умовах глобалізації : матеріали ІІ Міжнар. наук.-практ. конф. (Чернігів, 10 жовт. 2025 р.). Чернігів, 2025. С. 18–22. DOI: 10.64076/ihrc251010.05.
2. Alsalamah M., Alwabli H., Alqwifli H., Ibrahim D. M. A Review Study on SQL Injection Attacks, Prevention, and Detection. ISeCure: The ISC Int'l J. of Information Security. 2021. Vol. 13, No. 3. P. 1–9.
3. Patel N., Mohammed F., Soni S. SQL Injection Attacks: Techniques and Protection Mechanisms. International J. on Computer Science and Engineering. 2011. Vol. 3, No. 1. P. 199–203.
4. Федоренко А. А., Осадчий Б. І., Коржик В. В. Аналіз методів виявлення вразливостей web-ресурсів до SQL-ін'єкцій. Сучасний захист інформації. 2023. № 3(55). С. 57–61. DOI: 10.31673/2409-7292.2023.030008.
5. Замрій І. В., Шахматов І. О. Автоматизація оцінки безпеки вебзастосунків засобами Python. Зв'язок. 2025. № 4. С. 58–66.
6. Slotkienė A., Poška A., Stefanovič P., Ramanauskaitė S. Effect of Deep Recurrent Architectures on Code Vulnerability Detection: Performance Evaluation for SQL Injection in Python. Electronics. 2025. Vol. 14, No. 17. Art. 3436. DOI: 10.3390/electronics14173436.
7. Clarke J. SQL Injection Attacks and Defense. 2nd ed. Waltham : Syngress, 2012. 576 p.
8. SQL injection in 2025: AI, WAF & defense. URL: <https://www.sitewall.net/sql-injection-in-2025-ai-waf-defense/> (дата звернення: 03.02.2026).
9. The state of SQL injections. URL: <https://www.aikido.dev/blog/the-state-of-sql-injections> (дата звернення: 17.02.2026).

10. SQL injection risks, real-world examples and the role of Auxin Security. URL : <https://auxin.io/sql-injection-risks-real-world-examples-and-the-role-of-auxin-security/> (дата звернення: 26.02.2026).
11. #StopRansomware: CL0P Ransomware Gang Exploits CVE-2023-34362 MOVEit Vulnerability. CISA Cybersecurity Advisory AA23-158A. URL: <https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-158a> (дата звернення: 02.03.2026).
12. Земляний О. О. Система захисту інформації веб-застосунку з оренди нерухомості : магістер. дис. ... ступеня магістра : спец. 126 «Інформаційні системи та технології». Київ : КПІ ім. Ігоря Сікорського, 2021. 111 с.
13. Millions of user records stolen from 65 websites via SQL injection attacks. URL: <https://www.securityweek.com/millions-of-user-records-stolen-from-65-websites-via-sql-injection-attacks/> (дата звернення: 03.03.2026).
14. Саган О. А., Швирло К. Б. Аналіз SQL-ін'єкцій: типи атак та методи захисту. Актуальні задачі сучасних технологій : матеріали XIII Міжнар. наук.-практ. конф. молодих учених та студентів (Тернопіль, 11–12 груд. 2024 р.). Тернопіль, 2024. С. 484.
15. Django. QuerySet API reference. URL: <https://docs.djangoproject.com/en/stable/ref/models/queriesets/> (дата звернення: 10.03.2026).
16. Django. Form and field validation. URL: <https://docs.djangoproject.com/en/stable/ref/forms/validation/> (дата звернення: 05.03.2026).
17. Django. Built-in validators reference. URL: <https://docs.djangoproject.com/en/stable/ref/validators/> (дата звернення: 05.03.2025).
18. Django. Security in Django. URL: <https://docs.djangoproject.com/en/stable/topics/security/> (дата звернення: 05.03.2025).

- 19.Django. Middleware. URL:
<https://docs.djangoproject.com/en/stable/ref/middleware/> (дата звернення:
10.03.2025).
- 20.Каштан В. Ю., Іванов Д. В. Конспект лекцій з дисципліни «Бази даних в інформаційних системах». Частина 1: для студентів спец. 126 «Інформаційні системи та технології». Дніпро : НТУ «ДП», 2021. 58 с.
- 21.Сафонов В. Р. Система аудиту та аналізу безпеки вебдодатків : кваліф. робота на здобуття ступеня бакалавра : спец. 125 «Кібербезпека». Хмельницький : ХНУ, 2025. 84 с.
- 22.Сьомик Д. І. Автоматизована система аналізу XSS та CSRF вразливостей вебдодатків на Python : кваліф. робота на здобуття ступеня бакалавра : спец. 125 «Кібербезпека». Хмельницький : ХНУ, 2025. 90 с.

ДОДАТКИ

Додаток А. Результати функціонального тестування

№	Маршрут/сторінка	Метод	Результат
1	/users/login/	GET, POST (коректні дані)	HTTP 200 / Перехід на розклад
2	/users/login/ (неправильний пароль)	POST	HTTP 200, форма з помилкою
3	/users/list/ (адмін)	GET	HTTP 200, список користувачів
4	/users/list/ (студент)	GET	Перенаправлення на розклад
5	/users/create/	POST (валідні дані)	HTTP 302, створено користувача
6	/users/edit/<pk>/	POST (зміна ролі)	HTTP 302, дані оновлено
7	/users/profile/	GET	HTTP 200, профіль поточного користувача
8	/organization/faculties/	GET	HTTP 200, список факультетів
9	/organization/specialities/	GET	HTTP 200, список спеціальностей
10	/organization/departments/	GET	HTTP 200, список кафедр
11	/organization/disciplines/	GET	HTTP 200, список дисциплін
12	/schedule/	GET (адмін)	HTTP 200, повний розклад
13	/schedule/my/	GET (студент)	HTTP 200, розклад групи студента
14	/schedule/lesson/create/	POST (валідні дані)	HTTP 302, заняття додано
15	/schedule/lesson/<pk>/edit/	POST	HTTP 302, заняття оновлено
16	/schedule/lesson/<pk>/delete/	POST	HTTP 302, заняття видалено
17	/messaging/	GET	HTTP 200, вхідні повідомлення
18	/messaging/create/	POST (валідні дані)	HTTP 302, повідомлення надіслано
19	/users/logout/	POST	HTTP 302, вихід із системи

Додаток Б. Порівняння рівня безпеки до та після впровадження захисту

№	Вектор / аспект безпеки	До впровадження захисту	Після впровадження захисту
1	SQL-ін'єкція через форму (класична)	Конкатенація рядків дозволяє виконання довільного SQL	ORM параметризує запити автоматично
2	SQL-ін'єкція Union-based	UNION SELECT розкриває дані з інших таблиць	ORM та SQLInjectionValidator блокує UNION
3	Time-based blind ін'єкція	SLEEP(5) / WAITFOR DELAY дозволяє розвідку БД	SQLInjectionValidator виявляє SLEEP/WAITFOR
4	Обхід автентифікації	admin'-- пропускає перевірку пароля	ORM та валідатор блокують OR/AND маніпуляції
5	ORDER BY ін'єкція	sort=last_name;DROP TABLE users;-- виконується	sort_map whitelist відхиляє невідомі значення
6	DDL-атаки (DROP/ALTER TABLE)	Вразливий: обліковий запис БД має повні права	обліковий запис має лише SELECT/INSERT/UPDATE /DELETE
7	Розвідка схеми БД	INFORMATION_SCHEMA.TABLES розкриває схему БД	SQLInjectionValidator блокує INFORMATION_SCHEMA
8	Hex-кодована ін'єкція	0x41414141 обходить перевірки на рядки	SQLInjectionValidator розпізнає hex-шаблон
9	CSRF-атака	Виконується POST-запит з іншого домену	Django CsrfViewMiddleware відхиляє без токена (HTTP 403)
10	Несанкціонований доступ (IDOR)	Будь-який аутентифікований користувач бачить дані без обмежень	декоратори @login_required та @admin_required обмежують доступ
11	Слабкий пароль	Приймаються паролі «1234», «password»	PasswordComplexityValidator вимагає літеру, спецсимвол, мінімум 8 символів
12	Журналювання атак	Спроби SQL-ін'єкцій не фіксуються	security.log реєструє IP, час, тип підозрілого запиту

Додаток В. Оцінка ефективності реалізованих методів захисту

№	Метод захисту	Охоплені загрози	Результат тестування	Ризики, що залишилися
1	Django ORM (параметризовані запити)	Класичні, Union-based, stacked SQL-ін'єкції через поля форм	Жоден із 7 SQL-векторів не досяг рівня виконання запиту	ORDER BY та LIMIT не параметризуються засобами ORM
2	SQLInjectionValidator (8 регулярних виразів)	Логічні обходи, time-based blind, hex-кодування, DDL, UNION, schema enumeration	100 % виявлення (7/7 шкідливих) при 0 % хибних спрацьовувань (7/7 легітимних)	Нові вектори потребують ручного додавання патернів
3	sort_map whitelist (ORDER BY захист)	ORDER BY ін'єкції через GET-параметри сортування	Усі тестові значення поза whitelist замінені безпечним значенням за замовчуванням	Застосовується лише до параметра сортування
4	Мінімізація привілеїв БД (sslmode=require)	DDL-атаки, читання системних каталогів pg_catalog	Обліковий запис не має прав CREATE/DROP/ALTER — DDL-команди відхиляються БД	Захист діє лише при успішній ін'єкції; не усуває саму вразливість
5	CSRF-захист (CsrfViewMiddleware)	Cross-Site Request Forgery на всіх POST/PUT/DELETE ендпоінтах	Усі POST-запити без валідного токена отримали HTTP 403	Не захищає від XSS-атак, що можуть викрасти CSRF-токен
6	Рольова авторизація (@admin_required, @login_required)	IDOR, несанкціонований доступ, горизонтальне підвищення привілеїв	Повна ізоляція ролей: студент не отримав доступу до адмін-ресурсів	Не захищає від атак на рівні SQL при вразливостях в самому ORM

7	PasswordComplexityValidator	Слабкі та передбачувані паролі	6 з 6 недійсних паролів відхилено, 3 з 3 коректних прийнято	Не перевіряє словникові паролі (типу «Password1!»)
8	HTTPS, HSTS та Secure cookies	Man-in-the-Middle атаки, перехоплення сесії, session hijacking	Ввімкнено в production: SECURE_SSL_REDIRECT, HSTS 1 рік, HttpOnly та SameSite	Не активно в debug-режимі (тільки production)
9	Журналювання безпеки (security.log)	Brute-force, автоматизоване сканування, аномальна активність	Фіксація IP, мітки часу та типу підозрілих запитів	Фіксує атаку, а не запобігає їй
10	Secure session flags (SESSION_COOKIE_*)	Session hijacking, CSRF через cookies	Налаштовано HttpOnly=True, Secure=True, SameSite=Lax	Ефективний лише при увімкненому HTTPS

Додаток Г. Лістинг шаблону форми створення повідомлення з фільтрами отримувачів

```
{% extends 'base.html' %}
{% block title %}Нове повідомлення{% endblock %}
{% block content %}
<div class="row justify-content-center">
  <div class="col-md-7">
    <div class="card shadow-sm">
      <div class="card-body p-4">
        <h5 class="mb-3"><i class="bi bi-pencil-square"></i> Нове
повідомлення</h5>
        <form method="post">
          {% csrf_token %}

          {# — Отримувачі — #}
          <div class="mb-3">
            <label class="form-label fw-
semibold">Отримувачі</label>

            {# Фільтр за роллю #}
            <div class="d-flex align-items-center gap-2 mb-2">
              <span class="text-muted small">Роль:</span>
              <div class="btn-group btn-group-sm" id="role-
filter">
                <button type="button" class="btn btn-outline-
secondary active" data-role="all">Всі</button>
                <button type="button" class="btn btn-outline-
secondary" data-role="teacher">Викладачі</button>
                <button type="button" class="btn btn-outline-
secondary" data-role="student">Студенти</button>
              </div>
            </div>

            {# Фільтр за групою / курсом (лише для студентів) #}
            <div id="student-filters" class="d-none d-flex gap-2
mb-2">
              <select id="filter-group" class="form-select form-
select-sm" style="max-width:160px;">
                <option value="">Всі групи</option>
                {% for g in groups %}
                  <option value="{{ g.group_name }}">{{
g.group_name }}</option>
                {% endfor %}
              </select>
              <select id="filter-course" class="form-select form-
select-sm" style="max-width:130px;">
                <option value="">Всі курси</option>
                {% for c in courses %}
                  <option value="{{ c }}">{{ c }} курс</option>
                {% endfor %}
              </select>
            </div>
          </div>
        </form>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

```

</div>

{# Список отримувачів #}
<div class="border rounded p-2" style="max-
height:200px;overflow-y:auto;" id="recipient-list">
    {% for u in user_data %}
        <div class="recipient-item"
            data-role="{{ u.role }}"
            data-group="{{ u.group }}"
            data-course="{{ u.course }}">
            <label class="d-flex align-items-center gap-1">
                <input type="checkbox" name="recipients"
value="{{ u.id }}">
                    {{ u.name }}
                    {% if u.role == 'student' and u.group %}
                        <span class="text-muted small">({{ u.group }},
{{ u.course }} курс)</span>
                    {% endif %}
            </label>
        </div>
    {% empty %}
        <p class="text-muted small mb-0">Немає доступних
отримувачів.</p>
    {% endfor %}
</div>

<div id="no-recipients-msg" class="text-muted small
mt-1 d-none">
    Нікого не знайдено за обраними фільтрами.
</div>

{% if form.recipients.errors %}
    <div class="text-danger small">{{
form.recipients.errors }}</div>
{% endif %}
</div>

{# Тема #}
<div class="mb-3">
    <label class="form-label fw-semibold">{{
form.subject.label }}</label>
    {{ form.subject }}
    {% if form.subject.errors %}<div class="text-danger
small">{{ form.subject.errors }}</div>{% endif %}
</div>

{# Повідомлення #}
<div class="mb-3">
    <label class="form-label fw-semibold">{{
form.content.label }}</label>
    {{ form.content }}
    {% if form.content.errors %}<div class="text-danger
small">{{ form.content.errors }}</div>{% endif %}

```

```

        </div>

        <div class="d-flex gap-2">
            <button type="submit" class="btn btn-primary"><i
class="bi bi-send"></i> Надіслати</button>
            <a href="{% url 'messaging:inbox' %}" class="btn btn-
outline-secondary">Скасувати</a>
        </div>
    </form>
</div>
</div>
</div>
</div>

<script>
(function () {
    const roleButtons    = document.querySelectorAll('#role-filter
[data-role]');
    const studentFilters = document.getElementById('student-
filters');
    const filterGroup     = document.getElementById('filter-group');
    const filterCourse    = document.getElementById('filter-course');
    const items           = document.querySelectorAll('.recipient-
item');
    const noMsg           = document.getElementById('no-recipients-
msg');

    let currentRole = 'all';

    function applyFilters() {
        const group = filterGroup.value;
        const course = filterCourse.value;
        let visible = 0;

        items.forEach(item => {
            const role    = item.dataset.role;
            const iGroup  = item.dataset.group;
            const iCourse = item.dataset.course;

            let show = true;
            if (currentRole !== 'all' && role !== currentRole) show =
false;
            if (currentRole === 'student') {
                if (group && iGroup !== group) show = false;
                if (course && iCourse !== course) show = false;
            }
            item.style.display = show ? '' : 'none';
            if (show) visible++;
        });

        noMsg.classList.toggle('d-none', visible > 0);
    }
}

```

```

roleButtons.forEach(btn => {
  btn.addEventListener('click', () => {
    roleButtons.forEach(b => b.classList.remove('active'));
    btn.classList.add('active');
    currentRole = btn.dataset.role;
    studentFilters.classList.toggle('d-none', currentRole !==
'student');
    if (currentRole !== 'student') {
      filterGroup.value = '';
      filterCourse.value = '';
    }
    applyFilters();
  });
});

filterGroup.addEventListener('change', applyFilters);
filterCourse.addEventListener('change', applyFilters);
})();
</script>
{% endblock %}

```

Додаток Г. Лістинг класів-обробників запитів модуля управління розкладом

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.shortcuts import render, redirect, get_object_or_404
from django.views import View
from django.contrib import messages
from django.db import transaction

from .models import Lesson, Comment, Semester, Week, StudentGroup,
Classroom, Practice
from .forms import LessonForm, CommentForm, SemesterForm,
WeekForm, StudentGroupForm, ClassroomForm, PracticeForm

class ScheduleListView(LoginRequiredMixin, View):
    template_name = 'schedule/list.html'

    def get(self, request):
        semester_id = request.GET.get('semester')
        week_id = request.GET.get('week')
        semesters = Semester.objects.all()
        active_sem =
Semester.objects.filter(is_active=True).first()
        semester = get_object_or_404(Semester, pk=semester_id)
if semester_id else active_sem
        weeks = semester.weeks.all() if semester else
Week.objects.none()
        week = get_object_or_404(Week, pk=week_id) if
week_id else weeks.first()
        lessons =
Lesson.objects.filter(week=week).prefetch_related('groups',
'classrooms').select_related('teacher__user', 'discipline') if
week else Lesson.objects.none()
        return render(request, self.template_name, {
            'semesters': semesters,
            'semester': semester,
            'weeks': weeks,
            'week': week,
            'lessons': lessons,
            'days': Lesson.Day.choices,
        })

class LessonCreateView(LoginRequiredMixin, View):
    template_name = 'schedule/lesson_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request):
```

```

        return render(request, self.template_name, {'form':
LessonForm(), 'title': 'Додати заняття'})

    @transaction.atomic
    def post(self, request):
        form = LessonForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, 'Заняття додано.')
            return redirect('schedule:list')
        return render(request, self.template_name, {'form': form,
'title': 'Додати заняття'})

class LessonEditView(LoginRequiredMixin, View):
    template_name = 'schedule/lesson_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        lesson = get_object_or_404(Lesson, pk=pk)
        return render(request, self.template_name, {'form':
LessonForm(instance=lesson), 'title': 'Редагувати заняття',
'lesson': lesson})

    @transaction.atomic
    def post(self, request, pk):
        lesson = get_object_or_404(Lesson, pk=pk)
        form = LessonForm(request.POST, instance=lesson)
        if form.is_valid():
            form.save()
            messages.success(request, 'Заняття оновлено.')
            return redirect('schedule:list')
        return render(request, self.template_name, {'form': form,
'title': 'Редагувати заняття', 'lesson': lesson})

class LessonDeleteView(LoginRequiredMixin, View):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def post(self, request, pk):
        lesson = get_object_or_404(Lesson, pk=pk)
        lesson.delete()
        messages.success(request, 'Заняття видалено.')
        return redirect('schedule:list')

```

```

class CommentView(LoginRequiredMixin, View):
    template_name = 'schedule/comment_form.html'

    def get(self, request, lesson_pk):
        lesson = get_object_or_404(Lesson, pk=lesson_pk)
        comment = getattr(lesson, 'comment', None)
        form = CommentForm(instance=comment)
        return render(request, self.template_name, {'form': form,
'lesson': lesson})

    def post(self, request, lesson_pk):
        lesson = get_object_or_404(Lesson, pk=lesson_pk)
        comment = getattr(lesson, 'comment', None)
        form = CommentForm(request.POST, instance=comment)
        if form.is_valid():
            obj = form.save(commit=False)
            obj.lesson = lesson
            obj.author = request.user
            obj.save()
            messages.success(request, 'Коментар збережено.')
            return redirect('schedule:list')
        return render(request, self.template_name, {'form': form,
'lesson': lesson})

class CommentDeleteView(LoginRequiredMixin, View):
    def post(self, request, lesson_pk):
        lesson = get_object_or_404(Lesson, pk=lesson_pk)
        comment = get_object_or_404(Comment, lesson=lesson)
        if request.user.is_admin_role or comment.author ==
request.user:
            comment.delete()
            messages.success(request, 'Коментар видалено.')
            return redirect('schedule:list')

```

— Semesters

```

class SemesterListView(LoginRequiredMixin, View):
    template_name = 'schedule/semester_list.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'semesters':
Semester.objects.all()})

class SemesterCreateView(LoginRequiredMixin, View):

```



```

template_name = 'schedule/semester_form.html'

def dispatch(self, request, *args, **kwargs):
    if not request.user.is_admin_role:
        return redirect('schedule:list')
    return super().dispatch(request, *args, **kwargs)

def get(self, request):
    return render(request, self.template_name, {'form':
SemesterForm(), 'title': 'Додати семестр'})

def post(self, request):
    form = SemesterForm(request.POST)
    if form.is_valid():
        form.save()
        messages.success(request, 'Семестр додано.')
        return redirect('schedule:semester_list')
    return render(request, self.template_name, {'form': form,
'title': 'Додати семестр'})

class SemesterEditView(LoginRequiredMixin, View):
    template_name = 'schedule/semester_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        sem = get_object_or_404(Semester, pk=pk)
        return render(request, self.template_name, {'form':
SemesterForm(instance=sem), 'title': 'Редагувати семестр'})

    def post(self, request, pk):
        sem = get_object_or_404(Semester, pk=pk)
        form = SemesterForm(request.POST, instance=sem)
        if form.is_valid():
            form.save()
            messages.success(request, 'Семестр оновлено.')
            return redirect('schedule:semester_list')
        return render(request, self.template_name, {'form': form,
'title': 'Редагувати семестр'})

```

— Weeks

```

class WeekCreateView(LoginRequiredMixin, View):
    template_name = 'schedule/week_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:

```

```

        return redirect('schedule:list')
    return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'form':
WeekForm(), 'title': 'Додати тиждень'})

    def post(self, request):
        form = WeekForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, 'Тиждень додано.')
            return redirect('schedule:semester_list')
        return render(request, self.template_name, {'form': form,
'title': 'Додати тиждень'})

```

— Groups

```

class GroupListView(LoginRequiredMixin, View):
    template_name = 'schedule/group_list.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'groups':
StudentGroup.objects.select_related('speciality').all()})

class GroupCreateView(LoginRequiredMixin, View):
    template_name = 'schedule/group_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'form':
StudentGroupForm(), 'title': 'Додати групу'})

    def post(self, request):
        form = StudentGroupForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, 'Групу додано.')
            return redirect('schedule:group_list')
        return render(request, self.template_name, {'form': form,
'title': 'Додати групу'})

```

```

class GroupEditView(LoginRequiredMixin, View):
    template_name = 'schedule/group_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        group = get_object_or_404(StudentGroup, pk=pk)
        return render(request, self.template_name, {'form':
StudentGroupForm(instance=group), 'title': 'Редагувати групу'})

    def post(self, request, pk):
        group = get_object_or_404(StudentGroup, pk=pk)
        form = StudentGroupForm(request.POST, instance=group)
        if form.is_valid():
            form.save()
            messages.success(request, 'Групу оновлено.')
            return redirect('schedule:group_list')
        return render(request, self.template_name, {'form': form,
'title': 'Редагувати групу'})

```

— Classrooms

```

class ClassroomListView(LoginRequiredMixin, View):
    template_name = 'schedule/classroom_list.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'classrooms':
Classroom.objects.all()})

```

```

class ClassroomCreateView(LoginRequiredMixin, View):
    template_name = 'schedule/classroom_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'form':
ClassroomForm(), 'title': 'Додати аудиторію'})

```

```

    def post(self, request):
        form = ClassroomForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, 'Аудиторію додано.')
            return redirect('schedule:classroom_list')
        return render(request, self.template_name, {'form': form,
'title': 'Додати аудиторію'})

class ClassroomEditView(LoginRequiredMixin, View):
    template_name = 'schedule/classroom_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        classroom = get_object_or_404(Classroom, pk=pk)
        return render(request, self.template_name, {'form':
ClassroomForm(instance=classroom), 'title': 'Редагувати
аудиторію'})

    def post(self, request, pk):
        classroom = get_object_or_404(Classroom, pk=pk)
        form = ClassroomForm(request.POST,
instance=classroom)
        if form.is_valid():
            form.save()
            messages.success(request, 'Аудиторію оновлено.')
            return redirect('schedule:classroom_list')
        return render(request, self.template_name, {'form': form,
'title': 'Редагувати аудиторію'})

```

— Practices

```

class PracticeListView(LoginRequiredMixin, View):
    template_name = 'schedule/practice_list.html'

    def get(self, request):
        return render(request, self.template_name, {'practices':
Practice.objects.select_related('group').all()})

class PracticeCreateView(LoginRequiredMixin, View):
    template_name = 'schedule/practice_form.html'

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:

```

```

        return redirect('schedule:list')
    return super().dispatch(request, *args, **kwargs)

    def get(self, request):
        return render(request, self.template_name, {'form':
PracticeForm(), 'title': 'Додати практику'})

    def post(self, request):
        form = PracticeForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, 'Практику додано.')
            return redirect('schedule:practice_list')
        return render(request, self.template_name, {'form': form,
'title': 'Додати практику'})

class PracticeDeleteView(LoginRequiredMixin, View):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        practice = get_object_or_404(Practice, pk=pk)
        return render(request, 'schedule/confirm_delete.html', {
            'object': practice, 'cancel_url':
'schedule:practice_list'
        })

    def post(self, request, pk):
        get_object_or_404(Practice, pk=pk).delete()
        messages.success(request, 'Практику видалено.')
        return redirect('schedule:practice_list')

```

— Delete views

```

class SemesterDeleteView(LoginRequiredMixin, View):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        semester = get_object_or_404(Semester, pk=pk)
        return render(request, 'schedule/confirm_delete.html', {
            'object': semester, 'cancel_url':
'schedule:semester_list'
        })

    def post(self, request, pk):

```

```

        get_object_or_404(Semester, pk=pk).delete()
        messages.success(request, 'Семестр видалено.')
        return redirect('schedule:semester_list')

class GroupDeleteView(LoginRequiredMixin, View):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        group = get_object_or_404(StudentGroup, pk=pk)
        return render(request, 'schedule/confirm_delete.html', {
            'object': group, 'cancel_url': 'schedule:group_list'
        })

    def post(self, request, pk):
        get_object_or_404(StudentGroup, pk=pk).delete()
        messages.success(request, 'Групу видалено.')
        return redirect('schedule:group_list')

class ClassroomDeleteView(LoginRequiredMixin, View):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_admin_role:
            return redirect('schedule:list')
        return super().dispatch(request, *args, **kwargs)

    def get(self, request, pk):
        classroom = get_object_or_404(Classroom, pk=pk)
        return render(request, 'schedule/confirm_delete.html', {
            'object': classroom, 'cancel_url':
'schedule:classroom_list'
        })

    def post(self, request, pk):
        get_object_or_404(Classroom, pk=pk).delete()
        messages.success(request, 'Аудиторію видалено.')
        return redirect('schedule:classroom_list')

```

— Особистий розклад

```

class MyScheduleView(LoginRequiredMixin, View):
    """Особистий розклад для викладача (свої заняття) або студента
    (по групі)."""
    template_name = 'schedule/my_schedule.html'

    def get(self, request):
        semester_id = request.GET.get('semester')
        week_id     = request.GET.get('week')

```

```

        semesters = Semester.objects.all()
        active_sem =
Semester.objects.filter(is_active=True).first()
        semester = get_object_or_404(Semester, pk=semester_id)
if semester_id else active_sem
        weeks = semester.weeks.all() if semester else
Week.objects.none()
        week = get_object_or_404(Week, pk=week_id) if
week_id else weeks.first()

        lessons = Lesson.objects.none()
        if week:
            qs =
Lesson.objects.filter(week=week).prefetch_related('groups',
'classrooms').select_related('teacher__user', 'discipline')
            if request.user.is_teacher_role:
                try:
                    lessons =
qs.filter(teacher=request.user.teacher_profile)
                except Exception:
                    lessons = Lesson.objects.none()
            elif request.user.is_student_role:
                try:
                    group = request.user.student_profile.group
                    lessons = qs.filter(groups=group) if group
else Lesson.objects.none()
                except Exception:
                    lessons = Lesson.objects.none()
            else:
                lessons = qs # адмін бачить все

        return render(request, self.template_name, {
            'semesters': semesters,
            'semester': semester,
            'weeks': weeks,
            'week': week,
            'lessons': lessons,
            'days': Lesson.Day.choices,
        })

```

Додаток Д. лістинг моделі заняття застосування розкладу

```
from django.db import models
from django.core.exceptions import ValidationError

class Semester(models.Model):
    term_number =
models.PositiveSmallIntegerField(verbose_name='Номер семестру')
    start_date = models.DateField(verbose_name='Початок')
    end_date = models.DateField(verbose_name='Закінчення')
    is_active = models.BooleanField(default=False,
verbose_name='Поточний')

    class Meta:
        verbose_name = 'Семестр'
        verbose_name_plural = 'Семестри'
        ordering = ['-start_date']

    def clean(self):
        if self.start_date and self.end_date and self.start_date
>= self.end_date:
            raise ValidationError('Дата початку має бути раніше
дати закінчення.')

    def __str__(self):
        return f'Семестр {self.term_number}
({self.start_date.year})'

class Week(models.Model):
    semester = models.ForeignKey(Semester,
on_delete=models.CASCADE, related_name='weeks',
verbose_name='Семестр')
    week_number =
models.PositiveSmallIntegerField(verbose_name='Номер тижня')
    start_date = models.DateField(verbose_name='Початок тижня')
    end_date = models.DateField(verbose_name='Кінець тижня')

    class Meta:
        verbose_name = 'Тиждень'
        verbose_name_plural = 'Тижні'
        ordering = ['start_date']
        unique_together = ('semester', 'week_number')

    def __str__(self):
        return f'Тиждень {self.week_number} ({self.start_date} -
{self.end_date})'

class StudentGroup(models.Model):
    group_name = models.CharField(max_length=50,
```



```

verbose_name='Назва групи')
    group_course =
models.PositiveSmallIntegerField(verbose_name='Курс')
    student_count = models.PositiveSmallIntegerField(default=0,
verbose_name='К-сть студентів')
    speciality = models.ForeignKey('organization.Speciality',
on_delete=models.SET_NULL, null=True, blank=True,
related_name='groups', verbose_name='Спеціальність')

    class Meta:
        verbose_name = 'Навчальна група'
        verbose_name_plural = 'Навчальні групи'
        ordering = ['group_course', 'group_name']

    def __str__(self): return self.group_name

class Classroom(models.Model):
    class Type(models.TextChoices):
        LECTURE = 'lecture', 'Лекційна'
        PRACTICE = 'practice', 'Практична'
        LAB = 'lab', 'Лабораторна'
        COMPUTER = 'computer', 'Комп'ютерна'

    classroom_name = models.CharField(max_length=20,
verbose_name='Номер')
    building = models.CharField(max_length=100, blank=True,
verbose_name='Корпус')
    capacity =
models.PositiveSmallIntegerField(verbose_name='Місткість')
    classroom_type = models.CharField(max_length=10,
choices=Type.choices, verbose_name='Тип')

    class Meta:
        verbose_name = 'Аудиторія'
        verbose_name_plural = 'Аудиторії'
        ordering = ['classroom_name']

    def __str__(self):
        return f'{self.classroom_name}
({self.get_classroom_type_display()})'

class Practice(models.Model):
    class Type(models.TextChoices):
        EDUCATIONAL = 'educational', 'Навчальна'
        INDUSTRIAL = 'industrial', 'Виробнича'
        PRE_GRADUATION = 'pre_graduation', 'Переддипломна'

    practice_type = models.CharField(max_length=20,
choices=Type.choices, verbose_name='Тип практики')
    start_date = models.DateField(verbose_name='Початок')
    end_date = models.DateField(verbose_name='Закінчення')

```

```
        group = models.ForeignKey(StudentGroup,
on_delete=models.CASCADE, related_name='practices',
verbose_name='Група')
```

```
class Meta:
    verbose_name = 'Практика'
    verbose_name_plural = 'Практики'
```

```
def __str__(self):
    return f'{self.get_practice_type_display()} - {self.group}'
```

```
class Lesson(models.Model):
    class Type(models.TextChoices):
        LECTURE = 'lecture', 'Лекція'
        PRACTICE = 'practice', 'Практика'
        LAB = 'lab', 'Лабораторна'
        SEMINAR = 'seminar', 'Семінар'
```

```
class Day(models.IntegerChoices):
    MONDAY = 1, 'Понеділок'
    TUESDAY = 2, 'Вівторок'
    WEDNESDAY = 3, 'Середа'
    THURSDAY = 4, 'Четвер'
    FRIDAY = 5, "П'ятниця"
    SATURDAY = 6, 'Субота'
```

```
    lesson_type = models.CharField(max_length=10,
choices=Type.choices, verbose_name='Тип')
    day_of_week =
models.PositiveSmallIntegerField(choices=Day.choices,
verbose_name='День тижня')
    lesson_time = models.TimeField(verbose_name='Час початку')
    teacher = models.ForeignKey('users.Teacher',
on_delete=models.CASCADE, related_name='lessons',
verbose_name='Викладач')
    discipline = models.ForeignKey('organization.Discipline',
on_delete=models.CASCADE, related_name='lessons',
verbose_name='Дисципліна')
    semester = models.ForeignKey(Semester,
on_delete=models.CASCADE, related_name='lessons',
verbose_name='Семестр')
    week = models.ForeignKey(Week,
on_delete=models.CASCADE, related_name='lessons',
verbose_name='Тиждень')
    groups = models.ManyToManyField(StudentGroup,
through='LessonGroup', related_name='lessons',
verbose_name='Групи')
    classrooms = models.ManyToManyField(Classroom,
through='ClassroomLesson', related_name='lessons',
verbose_name='Аудиторії')
```

```

class Meta:
    verbose_name = 'Заняття'
    verbose_name_plural = 'Заняття'
    ordering = ['day_of_week', 'lesson_time']

    def __str__(self):
        return f'{self.get_lesson_type_display()} - {self.discipline} ({self.get_day_of_week_display()} {self.lesson_time})'

class LessonGroup(models.Model):
    lesson = models.ForeignKey(Lesson, on_delete=models.CASCADE)
    group = models.ForeignKey(StudentGroup, on_delete=models.CASCADE)

    class Meta:
        unique_together = ('lesson', 'group')

class ClassroomLesson(models.Model):
    lesson = models.ForeignKey(Lesson, on_delete=models.CASCADE)
    classroom = models.ForeignKey(Classroom, on_delete=models.CASCADE)

    class Meta:
        unique_together = ('lesson', 'classroom')

class Comment(models.Model):
    lesson = models.OneToOneField(Lesson, on_delete=models.CASCADE, related_name='comment', verbose_name='Заняття')
    author = models.ForeignKey('users.CustomUser', on_delete=models.CASCADE, related_name='comments', verbose_name='Автор')
    content = models.TextField(max_length=1000, verbose_name='Текст')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        verbose_name = 'Коментар'
        verbose_name_plural = 'Коментарі'

    def __str__(self): return f'Коментар до {self.lesson}'

```