Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування інституту, назва факультету)

Кафедра комп'ютерних систем та мереж
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалаврський
(ступінь вищої освіти)

на тему РОЗРОБКА ПРОГРАМНОГО КОМПЛЕКСУ МОНІТОРИНГУ СИСТЕМНИХ ПАРАМЕТРІВ ТА ОБЛІКУ КЛІЄНТСЬКИХ ЗВЕРНЕНЬ ДЛЯ СЕРВІСНОГО ЦЕНРУ

Виконав: студент(ка) 4 курсу, групи КНТ-521
Спеціальності 123 Комп'ютерна інженерія
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Комп'ютерна інженерія
АФАНАСЬЄВ І.І.
(прізвище та ініціали)
Керівник ГРУШКО С.С.
(ПРІЗВИЩЕ та ініціали)
Рецензент КОЗІНА Г.Л.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Інститут, факультет Комп`ютерних наук і технологій
Кафедра Комп'ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри КУДЕРМЕТОВ Р.К.

"14" квітня 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

АФАНАСЬЄВА Ігоря Івановича
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка програмного комплексу моніторингу системних параметрів та обліку клієнтських звернень для сервісного центру. Development of a software system for monitoring system parameters and tracking customer requests for a service center.

керівник проєкту (роботи) ГРУШКО Світлана Сергіївна, к.т.н., доцент ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від " 08 " квітня 2025 року № 151
2. Строк подання студентом проєкту (роботи) 01 06 2025 року
3. Вихідні дані до проєкту (роботи) існуючі методи та засоби моніторингу системних параметрів, технології обліку клієнтських звернень, бібліотека OSHI-core, засоби розробки програмних комплексів на Java, база даних для зберігання інформації.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз предметної області та існуючих рішень; 2) Обґрунтування вибору технологій; 3) Проєктування архітектури та бази даних; 4) Реалізація програмного комплексу; 5) Тестування та впровадження системи
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ГРУШКО С.С., к.т.н., доцент		
Нормоконтроль	ПОЛЬСЬКА О.В., старший викладач		

7. Дата видачі завдання “ 14 ” квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області та технічних вимог	18.04.2025	
2	Визначення технологій для реалізації системи	22.04.2025	
3	Проектування архітектури програмного комплексу	28.04.2025	
4	Розробка бази даних та API	05.05.2025	
5	Реалізація серверної частини (Spring Boot)	13.05.2025	
6	Розробка клієнтського застосунку (JavaFX)	20.05.2025	
7	Реалізація адмін-панелі для спеціалістів	25.06.2025	
8	Тестування (функціональне, навантажувальне, безпека)	28.06.2025	
9	Оформлення документації та графічних матеріалів	30.06.2025	
10	Захист дипломної роботи	05.06.2025	

Студент(ка) _____ АФАНАСЬЄВ І.І.
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ ГРУШКО С.С.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 126 с., 1 табл., 26 рис., 3 дод., 17 джерел.

РОЗРОБКА ПРОГРАМНОГО КОМПЛЕКСУ МОНІТОРИНГУ СИСТЕМНИХ ПАРАМЕТРІВ ТА ОБЛІКУ КЛІЄНТСЬКИХ ЗВЕРНЕНЬ ДЛЯ СЕРВІСНОГО ЦЕНТРУ

Об’єктом дослідження є інформаційні системи для моніторингу стану комп'ютерів та автоматизації обробки запитів користувачів.

Метою кваліфікаційної роботи є розробка програмного комплексу для збору системних параметрів комп’ютерів та обліку клієнтських звернень у сервісному центрі.

Предметом дослідження є методи та засоби збору системних параметрів комп’ютерів, генерації звітів у форматі PDF, а також способи реалізації ефективного обміну повідомленнями між користувачами та сервісним центром.

У ході роботи було проведено аналіз існуючих рішень, обґрунтовано вибір технологій та спроєктовано систему, що включає серверну частину на базі Spring Boot, а також клієнтський застосунок на JavaFX.

Результатом роботи є розробка програмного комплексу, який дозволяє збирати системні параметри комп'ютерів, формувати звіти у форматі PDF, обробляти звернення користувачів та автоматизувати взаємодію між клієнтами та сервісним центром.

ABSTRACT

Bachelor’s Thesis Explanatory Report: 126 pp., 1 table, 26 fig., 3 app., 17 refs.

DEVELOPMENT OF A SOFTWARE SUITE FOR SYSTEM-PARAMETER MONITORING AND CUSTOMER-REQUEST MANAGEMENT IN A SERVICE CENTER

The object of the research is information systems for monitoring computer health and automating user-request processing.

The aim of the thesis is to develop a software suite that collects system parameters of client computers and maintains customer requests in a service center.

The subject of the research comprises methods and tools for collecting system parameters, generating PDF reports, and implementing efficient message exchange between users and the service center.

The study includes an analysis of existing solutions, justification of the chosen technologies, and design of a system that consists of a Spring Boot back-end and a JavaFX desktop client.

The result is a fully functional software suite that gathers computer system parameters, generates PDF reports, processes user requests, and automates interaction between clients and the service center.

ЗМІСТ

ВСТУП

У сучасному світі автоматизація бізнес-процесів відіграє важливу роль у підвищенні ефективності підприємств. Одним із ключових аспектів у сфері технічного обслуговування є своєчасне отримання інформації про стан комп’ютерного обладнання. Це особливо актуально для сервісних центрів, які займаються діагностикою та усуненням несправностей комп’ютерної техніки.

На сьогодні існують різні рішення для збору інформації про системні параметри комп’ютера, однак багато з них є складними у впровадженні або вимагають значних ресурсів для роботи. Крім того, більшість подібних рішень не забезпечують зручної інтеграції з іншими модулями, такими як система обліку звернень користувачів або генерація звітів у форматі PDF.

Актуальність теми зумовлена потребою у простому та ефективному інструменті, який би дозволяв користувачам самостійно ініціювати збір системної інформації, описати проблему та надіслати запит до сервісного центру, де спеціаліст може одразу ознайомитися з технічними характеристиками обладнання. Таким чином, підвищується ефективність взаємодії між сторонами, скорочується час на первинну діагностику, що позитивно впливає на загальний рівень обслуговування.

Запропонований програмний комплекс орієнтований на реалізацію засобів для збору діагностичної інформації про комп’ютерну техніку, формування звітів та покращення взаємодії між клієнтами і фахівцями сервісного центру. Це дозволяє оптимізувати обробку звернень, пришвидшити виявлення можливих причин несправностей та підвищити якість технічного обслуговування.

Метою роботи є розробка програмного комплексу для збору системних параметрів комп’ютерів, генерації звітів у форматі PDF та обліку клієнтських звернень у сервісному центрі.

Об’єктом дослідження є інформаційні системи для оцінки стану комп’ютерного обладнання та автоматизації обробки запитів користувачів.

Предметом дослідження є методи та засоби збору системних параметрів комп’ютерів, генерації звітів у форматі PDF, а також способи ефективного обміну повідомленнями між клієнтами та сервісним центром.

Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати наявні рішення для збору технічної інформації про комп’ютери;

– обґрунтувати вибір технологій для реалізації програмного комплексу;

– розробити серверну частину системи з використанням Spring Boot та WebSocket;

– розробити клієнтський застосунок на JavaFX для користувачів і спеціалістів;

– реалізувати механізм збору системних параметрів та генерації PDF-звітів;

– розробити систему обробки клієнтських звернень;

– провести тестування розробленого програмного комплексу.

1 ТЕОРЕТИЧНІ ОСНОВИ СТВОРЕННЯ ПРОГРАМНИХ КОМПЛЕКСІВ ДЛЯ ЗБОРУ ДАНИХ І ОБРОБКИ КЛІЄНТСЬКИХ ЗВЕРНЕНЬ

1.1 Аналіз предметної області

Сучасна індустрія обслуговування комп’ютерної техніки стикається з дедалі більшими вимогами до ефективності обробки звернень користувачів та швидкої діагностики проблем. В умовах зростання кількості звернень до сервісних центрів, а також ускладнення апаратного та програмного забезпечення комп’ютерів, виникає потреба в автоматизованих інструментах, що забезпечують збирання, аналіз та передачу технічної інформації про стан систем користувачів.

Традиційно, при зверненні до сервісного центру користувач описує проблему словами, часто без надання точної технічної інформації. У таких випадках спеціалісти змушені витрачати додатковий час на діагностику пристрою, що може затримувати процес ремонту або обслуговування. Особливо це актуально у випадках віддаленого консультування або в роботі з корпоративними клієнтами, де простій обладнання може мати фінансові наслідки.

З іншого боку, більшість сучасних операційних систем дозволяють отримати розширену системну інформацію — від відомостей про процесор, оперативну пам’ять, накопичувачі, до стану температурних датчиків, завантаження системи та версій драйверів. Але самостійне отримання та структурування цієї інформації може бути складним для звичайного користувача. Крім того, навіть маючи повну інформацію, її передача спеціалісту у зручному вигляді потребує певної стандартизації.

Таким чином, виникає необхідність у створенні програмного комплексу, який:

– збирає системну інформацію про ПК користувача;

– формує структурований звіт у зручному форматі (наприклад, PDF);

– дозволяє користувачеві надіслати цей звіт разом із описом проблеми спеціалісту;

– забезпечує інтерактивну комунікацію між сторонами (наприклад, через чат);

– дає можливість спеціалісту оцінити можливу причину несправності, без фізичного огляду ПК.

Таке рішення сприятиме підвищенню якості обслуговування клієнтів, зменшенню навантаження на персонал сервісного центру, а також оптимізації часу на обробку звернень. Крім того, централізований облік звернень та автоматизований збір технічної інформації створюють передумови для подальшої аналітики та вдосконалення сервісу.

Таким чином, предметна область дипломного дослідження охоплює аспекти технічної діагностики, автоматизації обробки клієнтських звернень, збору системної інформації та організації взаємодії між користувачем і сервісним центром за допомогою сучасних програмних рішень.

1.2 Огляд методів збору системних параметрів комп'ютера

Системні параметри комп’ютера — це набір технічних характеристик, які описують стан апаратного та програмного забезпечення. До них належать:

– модель;

– частота процесора;

– обсяг оперативної пам’яті;

– інформація про жорсткі диски;

– відеокарту;

– температура компонентів;

– завантаження системи;

– встановлену операційну систему.

Автоматизований збір таких параметрів має велике значення для діагностики несправностей та моніторингу технічного стану пристрою.

1.2.1 Вбудовані інструменти операційної системи

Більшість операційних систем мають власні засоби перегляду системної інформації.

У Windows, для цього використовуються інструменти System Information (msinfo32), Task Manager, командний рядок (наприклад, systeminfo, wmic) або PowerShell (Get-ComputerInfo, Get-WmiObject), які дозволяють отримати докладну інформацію про систему .

У Linux-системах інформацію можна отримати за допомогою команд lshw, top, free, df, vmstat, uptime або шляхом читання файлів /proc (/proc/cpuinfo, /proc/meminfo тощо).

У macOS застосовуються команди system_profiler, top, sysctl, а також доступні вбудовані графічні утиліти для перегляду системної інформації.

Переваги використання вбудованих інструментів операційної системи полягають у наступному:

– не потребує додаткового програмного забезпечення;
– інформація надається напряму з ОС;

До недоліків такого підходу належать:

– неінтуїтивний доступ для пересічного користувача;
– результати неструктуровані, складні для автоматизованої обробки;
– ускладнена кросплатформеність (різні інструменти для кожної ОС).

1.2.2 Використання WMI (Windows Management Instrumentation)

WMI — технологія, що дозволяє програмно отримувати різноманітні дані про апаратне та програмне забезпечення в середовищі Windows. Доступ до WMI можливий через PowerShell, .NET, Java (через J-Interop, Waffle, або спеціальні бібліотеки) [17].

Переваги:

– велика кількість доступної інформації;
– можливість віддаленого доступу в корпоративних мережах.

Недоліки:

– обмежено Windows-платформою;

– може потребувати спеціальних дозволів;

– складність реалізації в Java-додатках без сторонніх бібліотек.

1.2.3 Низькорівневі нативні API та драйвери

Деякі програми використовують низькорівневий доступ до апаратних компонентів через драйвери (наприклад, SpeedFan, HWiNFO). У Java це практично неможливо без JNI (Java Native Interface) або JNA (Java Native Access) [7].

Переваги:

– максимально точні показники (наприклад, температура з датчиків).

Недоліки:

– залежність від ОС;

– необхідність встановлення додаткових драйверів або підписаних компонентів;

– можливі проблеми з безпекою та сумісністю.

1.2.4 Кросплатформені бібліотеки для збору системної інформації

Одним із найефективніших рішень у Java є використання бібліотеки OSHI (Operating System and Hardware Information) - Java-орієнтованої кросплатформеної бібліотеки, яка отримує системні дані через JNA [10].

OSHI-core дозволяє отримувати:

– інформацію про CPU, RAM, диски, мережу;

– навантаження системи;

– дані з датчиків температури (якщо підтримується системою);

– операційну систему та її версію;

– відомості про користувачів, процеси та сесії.

Переваги:

– працює на Windows, Linux, macOS без додаткових налаштувань;

– написана повністю на Java;

– не вимагає використання JNI/JNA вручну.

Недоліки:

– обмеженість в порівнянні з нативними засобами (менша точність деяких датчиків);

– потреба у регулярному оновленні бібліотеки через зміни в ОС.

Таким чином, для створення програмного забезпечення, яке має збирати системну інформацію в автоматичному режимі та працювати на Java, найоптимальнішим варіантом є використання бібліотеки OSHI-core. Вона дозволяє швидко інтегрувати збір інформації в десктоп-додаток, не вимагаючи складної конфігурації чи встановлення стороннього ПЗ користувачем.

1.3 Аналіз існуючих рішень для моніторингу системних параметрів

1.3.1 AIDA64 — комплексний інструмент діагностики

AIDA64 — це одна з найпопулярніших програм для детального аналізу апаратного та програмного забезпечення комп’ютера. Інтерфейс програми, як показано на рисунку 1.1, реалізовано у вигляді зручного ієрархічного дерева, що дозволяє швидко орієнтуватися серед великої кількості даних.

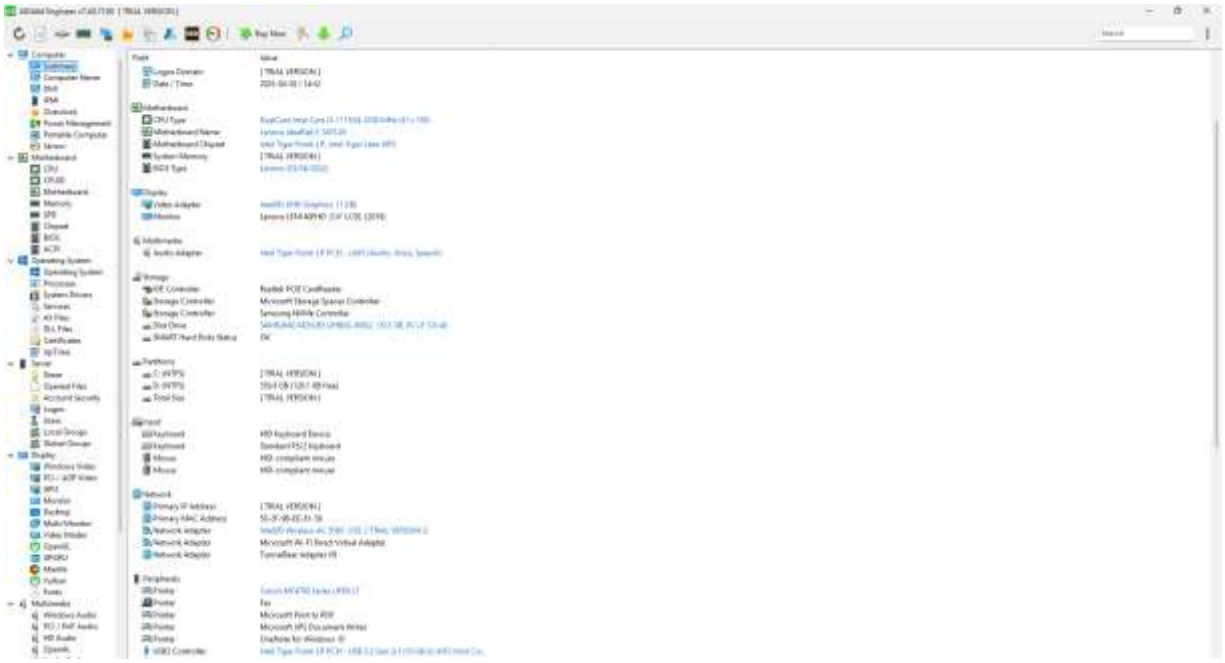


Рисунок 1.1 — Вікно AIDA64 з загальною інформацією про систему.

AIDA64 надає користувачу глибоку та структуровану інформацію про ключові компоненти системи: процесор, материнську плату, оперативну пам’ять, графічний адаптер, жорсткі диски, мережеві інтерфейси, а також показники температури, напруги та завантаження системи [1].

Переваги:

– висока деталізація та точність зібраної інформації;

– підтримка широкого спектру датчиків та обладнання;

– можливість експорту звітів у формати HTML, CSV, XML;

– інтерфейс українською, англійською та іншими мовами.

Недоліки:

– умовно-безкоштовна модель розповсюдження (безкоштовна версія має обмеження);

– відсутність повноцінного API для інтеграції в сторонні програми.

AIDA64 часто використовується системними адміністраторами, сервісними інженерами та ентузіастами для ручної діагностики та тестування обладнання. Проте, незважаючи на багатий функціонал, AIDA64 не є оптимальним вибором для створення кросплатформеного рішення. Закритість програмного забезпечення, ліцензійні обмеження, а також ускладнена інтеграція з Java-додатками обмежують її придатність для проєктів, що передбачають збір системної інформації без участі користувача. У контексті розробки програмного комплексу моніторингу для сервісного центру доцільніше застосовувати відкриті рішення, як-от OSHI-core.

1.3.2 HWiNFO

HWiNFO — безкоштовна програма для моніторингу системних параметрів у реальному часі. Забезпечує доступ до даних з апаратних сенсорів, підтримує логування та експортування даних [6].

Переваги:

– підтримка моніторингу температур, напруг, обертів вентиляторів;

– висока точність.

Недоліки:

– лише для Windows;

– відсутність відкритого API.

1.3.3 Speccy

Speccy — проста утиліта від розробників CCleaner. Надає базову інформацію про компоненти комп’ютера (процесор, RAM, жорсткі диски, ОС) [12].

Переваги:

– простота у використанні;

– компактний розмір.

Недоліки:

– обмежений функціонал;

– відсутність можливості експорту в структуровані формати;

– немає підтримки для автоматизації.

1.3.4 CPU-Z

CPU-Z — утиліта для отримання докладної інформації про центральний процесор, оперативну пам’ять і материнську плату [3].

Переваги:

– висока точність даних;

– швидка робота.

Недоліки:

– обмежене охоплення системних параметрів (не підтримує температурні сенсори, накопичувачі тощо);

– вузька спеціалізація.

1.4 Обґрунтування вибору технологій

Під час розробки програмного комплексу моніторингу системних параметрів і обліку клієнтських звернень для сервісного центру було обрано набір сучасних і перевірених технологій, які забезпечують стабільну роботу, розширюваність, кросплатформеність та зручність у підтримці. Оскільки розробка охоплює як

клієнтську, так і серверну частини, було проаналізовано відповідні стек-технології для обох рівнів.

1.4.1 Java як основна мова програмування

Вибір мови Java обумовлений її стабільністю, підтримкою об’єктно-орієнтованої парадигми, широкою екосистемою бібліотек і фреймворків, а також можливістю розробки як десктопних, так і серверних додатків. Java має багаторічну підтримку від спільноти й корпорацій, що гарантує надійність та довготривалу життєздатність проєкту.

1.4.2 JavaFX для створення графічного інтерфейсу користувача

JavaFX є потужною платформою для створення графічних інтерфейсів користувача в десктопних додатках. Це сучасна альтернатива Swing, що пропонує більш гнучкі можливості для побудови інтерфейсів. Однією з основних переваг JavaFX є підтримка таких технологій як FXML і CSS для побудови UI, що дозволяє створювати інтерфейси, які легко налаштовуються і підтримують сучасні дизайн-підходи [8].

Основні можливості JavaFX включають:

– модульна архітектура, що дозволяє розділяти функціонал інтерфейсу на окремі компоненти для зручнішої розробки та тестування;

– підтримка 2D і 3D графіки, що дозволяє створювати мультимедійні додатки з інтерактивними елементами;

– FXML для опису UI, що дозволяє розділяти логіку програми і опис інтерфейсу, що полегшує підтримку та модифікацію програми;

– вбудовані механізми анімації для створення плавних переходів і ефектів;

– кросплатформеність, підтримка Windows, Linux і macOS без змін у коді;

– JavaFX забезпечує повну інтеграцію з іншими компонентами стеку, такими як Spring, що дозволяє застосувати ін’єкцію залежностей і створити модульну структуру програми для ефективної роботи з користувачем та спеціалістами сервісного центру;

Це дозволяє реалізувати зручний інтерфейс для користувача та спеціаліста сервісного центру з підтримкою форм, звітів, повідомлень і чату.

1.4.3 Spring Framework на клієнтській частині

Для підвищення гнучкості, масштабованості та зручності розробки клієнтської частини було обрано використання Spring Framework (без Spring Boot), що забезпечує високий рівень контролю над конфігурацією додатку.

Ключовим компонентом є Spring Context, який реалізує шаблон інверсії управління (IoC) та підтримує ін’єкцію залежностей (DI) [14]. Це дозволяє легко керувати життєвим циклом об’єктів, автоматизувати створення зв'язків між компонентами та зменшити кількість "жорстко зв’язаного" коду.

Основні переваги використання Spring Framework на клієнтській частині:

– розділення відповідальностей за рахунок DI легко відокремлювати логіку UI, логіку роботи з даними та бізнес-логіку;

– тестованість - легко створювати unit-тести завдяки заміні залежностей через мок-об’єкти;

– гнучка конфігурація використання як XML-, так і Java-based конфігурацій для управління залежностями;

– модульність - структура клієнтського застосунку легко підтримується та розширюється за рахунок поділу на контейнери з окремими відповідальностями;

– інтеграція з JavaFX - завдяки DI можна легко підключати контролери, сервіси й менеджери станів без ручної ініціалізації.

Такий підхід дозволяє зробити клієнтську частину максимально гнучкою, незалежною та придатною для довготривалої підтримки.

1.4.4 Spring Boot для серверної частини

Серверна частина програмного комплексу реалізована на основі Spring Boot — фреймворку, який спрощує створення повноцінних веб-застосунків на базі Spring Framework шляхом автоматизації конфігурації, спрощеного запуску та інтеграції з іншими модулями [13].

Spring Boot забезпечує використання вбудованого вебсервера (Tomcat), що дозволяє запускати застосунок як автономний .jar, без потреби у сторонньому

вебсервері. Це значно прискорює процес розгортання та спрощує адміністрування системи.

Основні переваги Spring Boot:

– автоматична конфігурація компонентів на основі залежностей у classpath;

– швидкий запуск серверної частини з мінімальним налаштуванням;

– побудова REST API за допомогою зручного механізму контролерів із використанням анотацій (@RestController, @GetMapping, @PostMapping тощо);

– підтримка WebSocket — реалізована за допомогою модуля spring-websocket з можливістю використання протоколу STOMP;

– безпека — легко інтегрується з Spring Security для організації автентифікації та авторизації користувачів;

– взаємодія з базою даних через Spring Data JPA, що дозволяє працювати з PostgreSQL за допомогою об'єктно-реляційного відображення (ORM) [5];

– розширені можливості моніторингу та логування — підтримка модуля Spring Boot Actuator для збору метрик і спостереження за станом системи.

У межах проєкту за допомогою Spring Boot реалізується логіка:

– прийому звернень користувачів через REST-запити;

– обробки й збереження згенерованих PDF-звітів;

– маршрутизації повідомлень у WebSocket-чаті між користувачем і спеціалістом;

– взаємодії з базою даних для зберігання інформації про користувачів, звернення, історію повідомлень та стан обробки заявок.

1.4.5 WebSocket для реалізації онлайн-чату

Для реалізації чату між користувачем і спеціалістом застосовано технологію WebSocket.

На відміну від HTTP, WebSocket дозволяє підтримувати постійне двостороннє з’єднання, що дає змогу:

– миттєво обмінюватися повідомленнями;

– зменшити затримки;

– знизити навантаження на сервер порівняно з періодичним опитуванням (polling).

У Spring Boot WebSocket реалізується через модуль spring-websocket із підтримкою STOMP-протоколу [16].

1.4.6 PostgreSQL як система керування базами даних

У якості СКБД обрано PostgreSQL — надійне, розширюване рішення з відкритим кодом [11].

Його основні переваги:

– підтримка транзакцій, складних SQL-запитів, індексів;

– можливість зберігання JSON-структур;

– сумісність зі Spring Data JPA;

– активна спільнота та широка документація.

База даних містить інформацію про користувачів, заявки, системні звіти, повідомлення чату.

1.4.7 OSHI-core для збору системної інформації

Для реалізації функціоналу моніторингу системних параметрів у клієнтському застосунку було обрано бібліотеку OSHI (Operating System and Hardware Information) — кросплатформене рішення, що забезпечує збирання інформації про апаратні та програмні характеристики комп’ютера.

OSHI є бібліотекою з відкритим кодом, написаною мовою Java, яка використовує технологію JNA (Java Native Access) для доступу до нативних функцій операційної системи без необхідності написання JNI-коду або встановлення сторонніх драйверів. Це значно спрощує її інтеграцію в Java-додатки, особливо з урахуванням вимог кросплатформеності.

До основних можливостей бібліотеки OSHI належать:

– збір інформації про центральний процесор (CPU) — кількість ядер, тактова частота, завантаження, кеші;

– моніторинг оперативної пам’яті — обсяг, зайнятий/вільний обсяг, використання swap;

– інформація про накопичувачі — назви дисків, файлові системи, обсяги, використання;

– вивід параметрів мережевих інтерфейсів — MAC-адреси, IP-адреси, статистика передавання/отримання даних;

– дані про процеси та користувачів, завантажених у систему;

– інформація про температуру компонентів, якщо її підтримує материнська плата або відповідні датчики;

– зчитування загальної інформації про операційну систему, тип архітектури, час роботи системи тощо.

Бібліотека не потребує додаткових прав адміністратора або спеціального налаштування середовища, що робить її зручною у використанні навіть у середовищах із обмеженим доступом. Її модульна структура дає змогу легко отримати лише ту інформацію, яка необхідна для формування звіту, що особливо актуально для десктопного застосунку з обмеженими ресурсами.

Переваги використання OSHI-core у дипломному проєкті:

– кросплатформеність (підтримка Windows, Linux, macOS без зміни коду);

– відкритий код та активна спільнота розробників;

– простота інтеграції (додавання до проєкту через Maven/Gradle);

– легка адаптація до потреб користувача (можливість вибору типів параметрів, які треба виводити);

– відсутність зовнішніх залежностей, що позитивно впливає на стабільність роботи програми.

Таким чином, OSHI-core є оптимальним вибором для реалізації модулю збору системної інформації в контексті розробки універсального, надійного клієнтського застосунку для сервісного центру.

1.4.8 Генерація PDF-звітів

Для створення звітів у форматі PDF у програмному комплексі використовується бібліотека Apache PDFBox [2]. Вона є менш складною у використанні та надає всі необхідні можливості для роботи з PDF-документами,

дозволяючи створювати прості звіти, додавати текст, зображення та таблиці. Apache PDFBox підходить для проектів, де важлива проста генерація документів без потреби в складних функціях, таких як шифрування чи цифрові підписи.

PDF є універсальним форматом для збереження та обміну документами, оскільки забезпечує однакове відображення на різних платформах.

Основні переваги PDF:

– стабільність формату (PDF зберігає структуру документа, не змінюючись на різних пристроях чи ОС);

– зручність у зберіганні та архівації (PDF-документи легко передавати та зберігати на будь-яких пристроях, включаючи мобільні телефони);

– підтримка багатосторінкових документів (PDF дозволяє створювати документи з кількома сторінками та коректною нумерацією);

– збереження структури та стилю (PDF зберігає шрифти, таблиці, зображення та інші елементи форматування).

Ці переваги роблять PDF ідеальним вибором для звітів у програмному комплексі.

1.4.9 Використання JWT для забезпечення безпечної авторизації

З метою реалізації безпечного механізму доступу до функціоналу серверної частини планується використання JWT (JSON Web Token) — сучасного стандарту, що дозволяє реалізувати аутентифікацію та авторизацію без використання сесій [9].

JWT — це цифровий токен, який містить інформацію про користувача (ID, роль, термін дії тощо) в зашифрованому вигляді. Після успішної автентифікації сервер видає цей токен, який клієнт передає у кожному запиті в заголовку Authorization. Сервер перевіряє дійсність токена і, залежно від ролі користувача, надає доступ до відповідних ресурсів або функцій.

Основні переваги використання JWT:

– stateless-архітектура — відсутність потреби зберігати сесію на сервері;

– масштабованість — зручно використовувати в кластеризованих системах або мікросервісній архітектурі;

– безпека — можливість встановлення строку дії, шифрування та перевірки підпису токена;

– гнучкість доступу — реалізація доступу на основі ролей, наприклад, "користувач" та "спеціаліст".

У Spring Boot реалізація JWT здійснюється за допомогою Spring Security та спеціального фільтра (OncePerRequestFilter), який перевіряє валідність токена до обробки запиту [15]. У разі недійсного або відсутнього токена доступ до захищених ресурсів блокується.

Таким чином, використання JWT у поєднанні зі Spring Boot дозволяє створити безпечну, гнучку та масштабовану систему автентифікації та авторизації, яка відповідає сучасним вимогам до інформаційної безпеки.

1.5 Особливості використання OSHI-core

У межах розробки програмного комплексу для моніторингу системних параметрів комп’ютера важливу роль відіграє модуль збору інформації про апаратне та програмне забезпечення. Для цієї мети було обрано бібліотеку OSHI-core (Operating System and Hardware Information) — потужний засіб для кросплатформеного збору системної інформації мовою Java. Вона базується на механізмі Java Native Access (JNA), що забезпечує доступ до нативних функцій операційної системи без потреби написання коду на C/C++ або використання JNI (Java Native Interface).

OSHI надає уніфікований API для отримання даних про апаратні компоненти (процесор, пам’ять, накопичувачі, мережу, сенсори) та програмне середовище (версію ОС, аптайм, активні процеси, користувачів). Така абстракція спрощує розробку та підвищує портативність застосунків, особливо у мультиплатформених середовищах (див. рисунок 1.2).

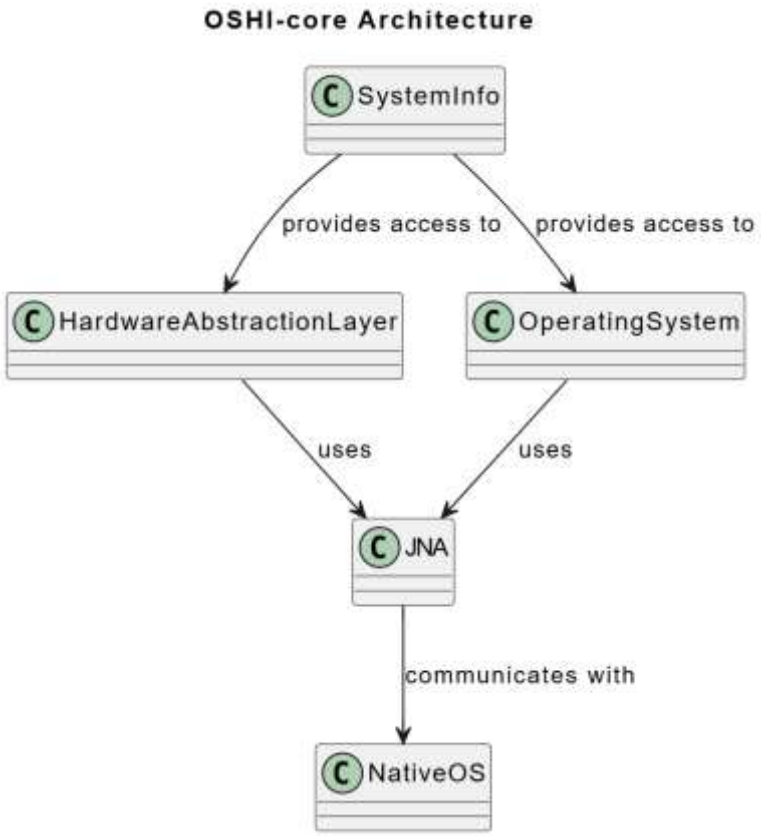


Рисунок 1.2 — Архітектура бібліотеки OSHI

1.5.1 Архітектура та принцип роботи

Бібліотека OSHI реалізована за принципами об’єктно-орієнтованого програмування та інкапсуляції доступу до системних ресурсів.

Вона не взаємодіє безпосередньо з апаратним забезпеченням, а отримує інформацію з відповідних джерел операційної системи, які є специфічними для кожної платформи.

Зокрема, у середовищі Windows використовується WMI (Windows Management Instrumentation), доступ до реєстру (зокрема, до гілки HKEY_PERFORMANCE_DATA), а також системи лічильників продуктивності (Performance Counters).

У Linux-системах дані зчитуються з віртуальних файлових систем /proc та /sys, а також за допомогою стандартних системних утиліт, таких як lscpu, lsblk, df, sensors тощо.

Для операційної системи macOS застосовуються такі інструменти, як sysctl, ioreg, top та ps.

У BSD-системах використовується аналогічний підхід, зокрема через інтерфейс sysctl та доступ до системних журналів.

Цей підхід забезпечує високу адаптивність бібліотеки і дозволяє отримувати максимально повні та точні дані без потреби в зовнішніх бінарних утилітах або запуску коду з підвищеними правами доступу.

Основні класи та інтерфейси OSHI:

– SystemInfo — головний вхідний клас, з якого починається робота з бібліотекою. Ініціалізує доступ до підсистем;

– HardwareAbstractionLayer (HAL) — шар абстракції для доступу до інформації про апаратне забезпечення;

– OperatingSystem — об’єкт для роботи з відомостями про операційну систему (назва та версія, аптайм, процеси, користувачі, файлові системи).

Усі дані за замовчуванням зчитуються без кешування, що гарантує актуальність отриманої інформації. Водночас багато об’єктів, таких як CentralProcessor, підтримують власне кешування результатів для зменшення навантаження при повторних запитах (наприклад, метод getSystemCpuLoadBetweenTicks() використовує збережені показники попередніх викликів для обчислення завантаження ЦП).

1.5.2 Основні можливості бібліотеки

За допомогою OSHI можна зібрати такі параметри:

– процесор (CPU) — модель, кількість ядер, навантаження, тактова частота;

– оперативна пам’ять (RAM) — загальний і доступний обсяг пам’яті, використання (див. рисунок 1.3);

– диски — об’єм, файлові системи, завантаження, серійні номери;

– мережева активність — активні інтерфейси, MAC-адреси, IP, швидкість з’єднання, обсяг переданих/отриманих даних;

– температура та датчики — температура процесора, напруга, оберти вентиляторів (за підтримки обладнання);

– операційна система — назва ОС, версія, архітектура, аптайм;

– користувачі та процеси — поточні користувачі, активні процеси, PID, використання ресурсів.

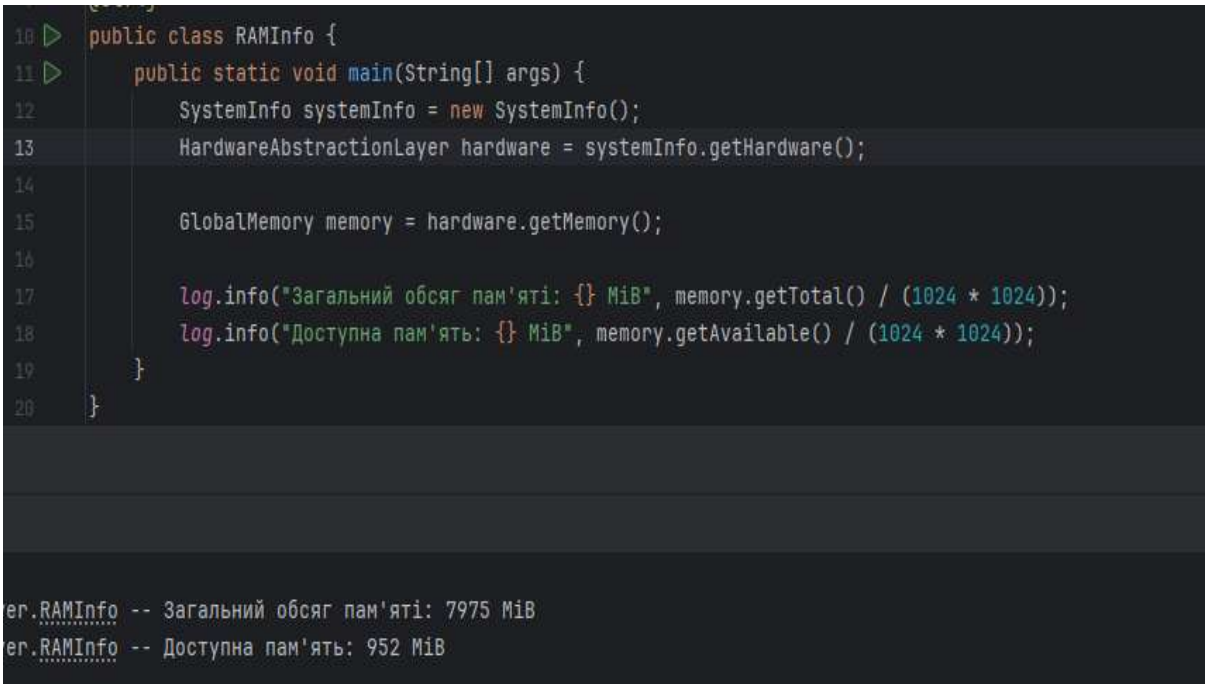


Рисунок 1.3 — Приклад отримання обсягу оперативної пам’яті

1.5.3 Переваги використання OSHI-core

Основні причини вибору саме OSHI:

– кросплатформеність — працює на більшості поширених ОС без зміни коду;

– відкритий код — проєкт підтримується спільнотою, активно оновлюється;

– легкість у використанні — зрозумілий API, зручна документація;

– безпека — не потребує рутових прав чи нативних модулів;

– продуктивність — оптимізовані виклики для мінімального навантаження.

1.5.4 Обмеження та недоліки

Попри численні переваги, OSHI має певні обмеження:

– точність і повнота зібраної інформації залежить від ОС та обладнання;

– обмежена підтримка деяких специфічних датчиків (температури, напруги);

– деякі методи можуть повертати null або 0 у разі відсутності підтримки драйверами.

У межах поставленого завдання функціоналу OSHI-core є повністю достатньо для реалізації цілей системного моніторингу користувацьких ПК.

1.6 Висновки до розділу 1

У ході виконання теоретичних досліджень було охоплено основні аспекти створення програмного комплексу для моніторингу системних параметрів комп'ютера та обліку клієнтських звернень.

Досягнуті наступні результати:

– проведено аналіз предметної області, що підтвердив актуальність створення програмного забезпечення для підвищення ефективності технічної підтримки;

– розглянуто сучасні методи та засоби збору інформації про апаратне та програмне забезпечення комп’ютера, зокрема через використання Java-бібліотеки OSHI-core;

– проаналізовано існуючі рішення у галузі системного моніторингу та клієнтського обслуговування, зокрема їх переваги та недоліки;

– обґрунтовано вибір технологій для розробки системи, таких як JavaFX, Spring Framework, Spring Boot, WebSocket, PostgreSQL та інструменти для генерації PDF-звітів;

– визначено архітектурні особливості програмного комплексу та сформовано загальну структуру взаємодії клієнтської та серверної частин.

Ці результати складають основу для подальшої розробки та реалізації програмного комплексу.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ

2.1 Функціональні та нефункціональні вимоги до системи

У цьому підрозділі наведено основні функціональні та нефункціональні вимоги до програмного комплексу моніторингу системних параметрів і обліку клієнтських звернень.

2.1.1 Функціональні вимоги

Функціональні вимоги визначають основні можливості системи, що мають бути реалізовані для забезпечення роботи програмного комплексу моніторингу системних параметрів та обліку клієнтських звернень для сервісного центру.

Загальні вимоги:

– система повинна підтримувати три типи користувачів (адміністратора, спеціаліста, клієнта);

– для доступу до функціоналу, що потребує авторизації, має бути реалізований механізм автентифікації та авторизації користувачів;

– користувацькі ролі мають бути реалізовані з відповідними рівнями доступу до функцій системи.

Вимоги до адміністратора:

– автоматичне створення облікового запису адміністратора при першому запуску серверної частини;

– можливість входу в систему за допомогою електронної пошти та паролю;

– можливість перегляду, редагування особистого профілю (ім’я, e-mail, пароль) з підтвердженням коду з e-mail;

– перегляд списку спеціалістів сервісного центру;

– додавання нового спеціаліста (введення ім’я, e-mail, пароль);

– видалення спеціаліста зі списку;

Вимоги до спеціаліста:

– можливість входу в систему за допомогою електронної пошти та паролю (аккаунт створюється адміністратором);

– можливість перегляду, редагування особистого профілю (ім’я, e-mail, пароль) з підтвердженням коду з e-mail;

– перегляд кількості відкритих звернень зі статусом «Open», де ще не призначено відповідального спеціаліста;

– перегляд кількості звернень у статусі «In Progress», де користувач призначений відповідальним спеціалістом;

– доступ до повного списку звернень, до яких спеціаліст має доступ;

– відкриття чату звернення, перегляд і надсилання повідомлень та можливість завантаження звіту прикріпленого клієнтом;

– автоматичне призначення себе відповідальним спеціалістом при першому відкритті звернення зі статусом «Open» (переведення звернення у статус «In Progress»);

– можливість закривати звернення (переведення у статус «Closed»).

Вимоги до клієнта:

– реєстрація з підтвердженням e-mail (надсилання коду підтвердження);

– можливість входу в систему за допомогою електронної пошти та паролю;

– можливість перегляду, редагування особистого профілю (ім’я, e-mail, пароль) з підтвердженням коду з e-mail;

– генерація звіту про системні параметри комп’ютера;

– завантаження згенерованого звіту;

– перегляд власних звітів;

– видалення звітів (тільки звіти, не прикріплені до звернень)

– створення нового звернення до сервісного центру (із зазначенням теми, опису та автоматичним прикріпленням звіту, якщо згенерований);

– перегляд списку чатів своїх звернень

– відкриття чату звернення, обмін повідомленнями зі спеціалістом;

– можливість закривати звернення (зміна статусу на «Closed»).

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають обмеження та характеристики якості системи, які мають бути дотримані під час розробки, впровадження та експлуатації програмного комплексу.

Вимоги до продуктивності:

– система повинна забезпечувати обробку не менше 50 одночасних користувачів без втрати продуктивності;

– час відгуку інтерфейсу клієнтської частини не повинен перевищувати 1 секунди при стандартних операціях (наприклад, перегляд списку звернень, відправка повідомлення в чаті).

Вимоги до безпеки:

– для доступу до захищених ресурсів має бути реалізований механізм автентифікації та авторизації користувачів;

– персональні дані користувачів повинні зберігатися у зашифрованому вигляді (наприклад, паролі — із використанням алгоритму bcrypt);

– система повинна бути захищена від SQL-ін’єкцій.

Вимоги до зручності використання (usability):

– клієнтський інтерфейс повинен бути інтуїтивно зрозумілим, підтримувати українську мову;

– клієнтський інтерфейс повинен бути інтуїтивно зрозумілим, підтримувати українську мову;

Вимоги до надійності:

– у разі збою під час виконання критичних операцій користувач має отримати інформативне повідомлення про помилку;

– система повинна мати механізми журналювання помилок та подій.

Вимоги до масштабованості та підтримки:

– програмний комплекс повинен мати модульну архітектуру, яка дозволяє розширювати функціонал (наприклад, додавання нових ролей або модулів);

– підтримка оновлення компонентів без втрати даних користувачів;

– можливість розгортання серверної частини на ОС Linux.

2.2 Архітектура програмного комплексу

У цьому підрозділі наведено архітектурне рішення для програмного комплексу моніторингу системних параметрів та обліку клієнтських звернень для сервісного центру. Архітектура системи побудована на основі клієнт-серверної моделі, що забезпечує ефективний розподіл функціоналу між клієнтською та серверною частинами та дозволяє масштабувати та розширювати систему відповідно до зростаючих потреб.

Програмний комплекс складається з трьох основних компонентів:

– клієнтська частина;

– серверна частина;

– база даних.

Клієнтська частина — це десктопний застосунок, розроблений із використанням JavaFX та Spring Core. Вона забезпечує взаємодію користувача з системою, включаючи збір системних параметрів комп’ютера за допомогою бібліотеки OSHI-core, формування та надсилання звернень, перегляд і редагування профілю, а також обмін повідомленнями зі спеціалістами сервісного центру через чат.

Серверна частина представлена застосунком на основі Spring Boot, який реалізує основну бізнес-логіку системи, обробку REST-запитів від клієнтів,

зберігання даних у реляційній базі, генерацію PDF-звітів на основі бібліотеки Apache PDFBox, а також надсилання електронних листів користувачам (через JavaMail) для підтвердження дій.

База даних, що використовується у системі, — це PostgreSQL. Вона призначена для зберігання інформації про користувачів, звернення, звіти, повідомлення чату та інші об’єкти, які забезпечують роботу програмного комплексу.

Взаємодія між клієнтською та серверною частинами здійснюється за допомогою HTTP-запитів до REST API, що забезпечує основний функціонал системи. Для реалізації чату між клієнтом і спеціалістом використовується WebSocket API, що забезпечує обмін повідомленнями в реальному часі.

Схему архітектури програмного комплексу наведено на рисунку 2.1.

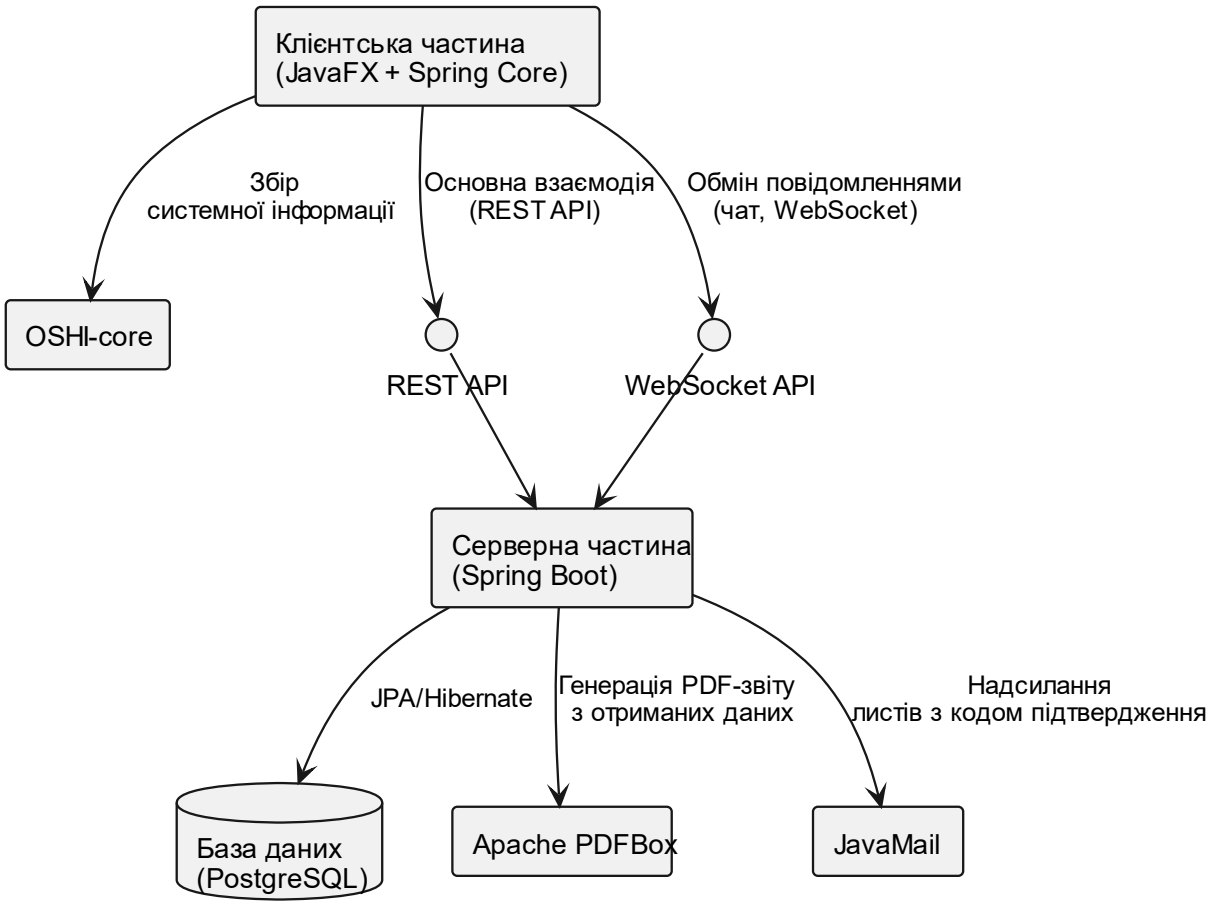


Рисунок 2.1 – Діаграма компонентів архітектури програмного комплексу моніторингу системних параметрів

Обрана архітектурна модель дозволяє:

– розділити відповідальність між клієнтською та серверною частинами для підвищення безпеки, зручності розширення й супроводу системи;

– реалізувати багаторівневу обробку даних та централізоване зберігання інформації;

– забезпечити масштабованість та можливість подальшого розвитку програмного комплексу, зокрема, додавання нових ролей або модулів;

– спростити інтеграцію з іншими сервісами або зовнішніми системами в майбутньому.

На діаграмі відображено основні компоненти системи та їх взаємозв’язки. Клієнтська частина взаємодіє із серверною частиною через REST API для виконання основних операцій та через WebSocket API для обміну повідомленнями у чаті. Серверна частина обробляє запити клієнтів, генерує PDF-звіти, надсилає електронні листи та взаємодіє з базою даних для збереження й обробки інформації. Збір системних параметрів комп’ютера здійснюється на клієнті за допомогою бібліотеки OSHI-core, а формування звітів у форматі PDF – на сервері за допомогою Apache PDFBox.

2.3 Проектування бази даних

Важливим етапом розробки програмного комплексу є проектування бази даних, яка забезпечує надійне зберігання, цілісність і структуровану обробку всієї інформації, необхідної для роботи системи моніторингу системних параметрів та обліку клієнтських звернень. Для реалізації програмного комплексу було обрано реляційну модель бази даних, що дозволяє гнучко реалізувати всі необхідні функціональні сценарії, забезпечити ефективне виконання запитів і масштабованість системи. В якості системи керування базами даних використовується PostgreSQL.

2.3.1 Структура бази даних

Логічна структура бази даних програмного комплексу передбачає наявність чотирьох основних сутностей, що відображають ключові об’єкти предметної області.

Таблиця користувачів (APP_USER) зберігає інформацію про всіх користувачів системи. У ній містяться такі атрибути, як унікальний ідентифікатор користувача, електронна пошта, пароль у захищеному вигляді, роль користувача (адміністратор, клієнт або спеціаліст), а також, за необхідності, відображуване ім’я (nickname).

Таблиця звернень (ISSUE) містить дані про всі клієнтські звернення до сервісного центру. Для кожного звернення зберігається унікальний ідентифікатор, ідентифікатор клієнта (користувача), тема звернення, опис проблеми, статус (open, in_progress, closed), дата та час створення, ідентифікатор призначеного спеціаліста (за наявності), а також можливий зв’язок із прикріпленим звітом.

Таблиця повідомлень (MESSAGE) призначена для зберігання історії спілкування між клієнтом та спеціалістом у межах конкретного звернення. Вона містить ідентифікатор повідомлення, текстове наповнення, ідентифікатор автора (користувача), час відправлення, а також може містити зв’язок із зверненням та прикріпленим звітом.

Таблиця звітів (REPORT) використовується для зберігання всіх сформованих системних звітів, зокрема звітів про системні параметри комп’ютера. Для кожного звіту фіксується унікальний ідентифікатор, ім’я файлу, вміст файлу у вигляді BLOB, ідентифікатор користувача, для якого створено звіт, а також дата створення.

Зовнішні ключі забезпечують логічні зв’язки між таблицями, зокрема зв’язок між зверненням та його клієнтом і спеціалістом, між повідомленням та зверненням або звітом, між звітом і користувачем.

2.3.2 Схема даних та взаємозв’язки між таблицями

На рисунку 2.2 представлено схему даних (ERD), що відображає основні сутності бази даних та взаємозв’язки між ними.

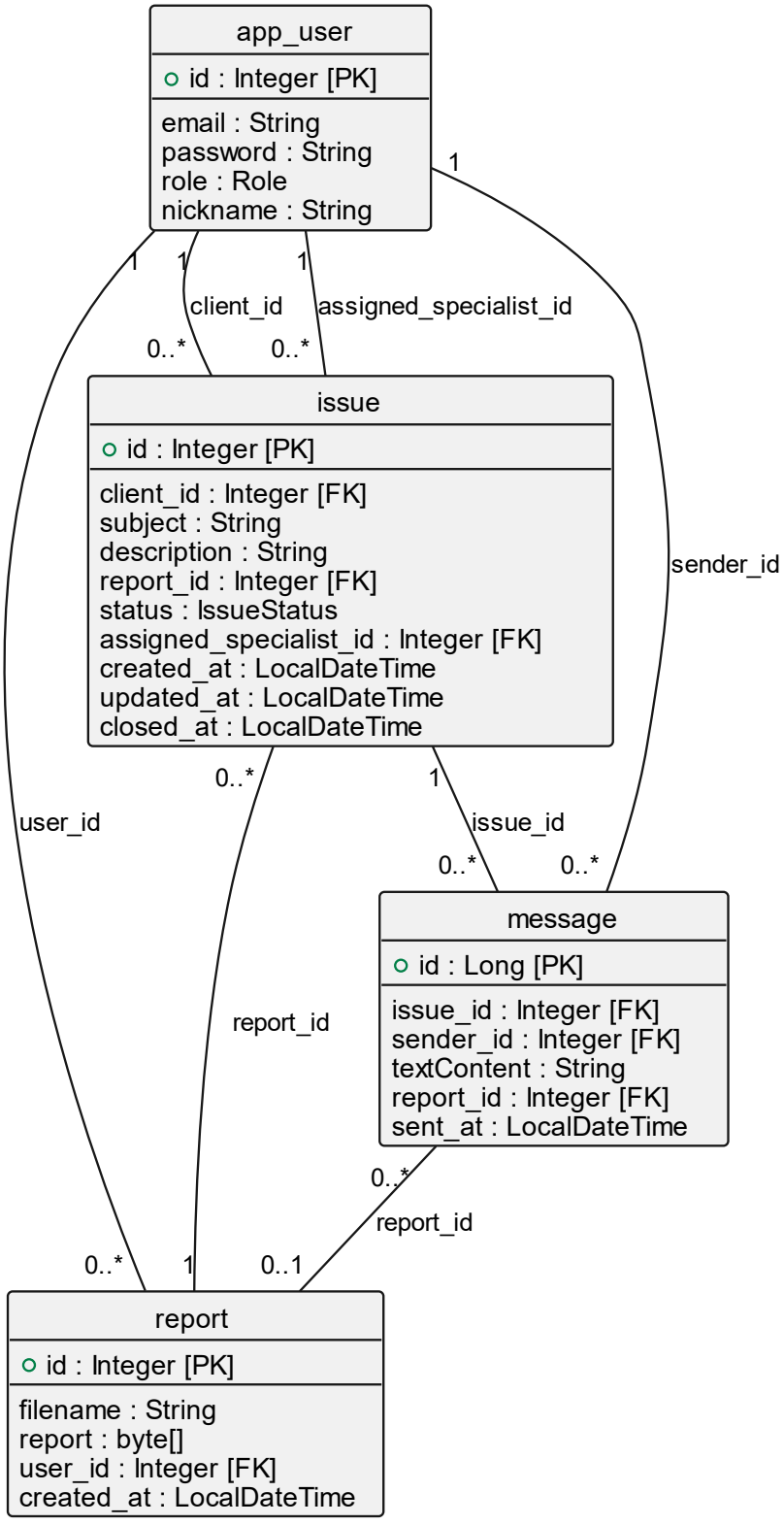


Рисунок 2.2 – Схема даних та взаємозв’язки між таблицями бази даних

Як видно з діаграми, між таблицями реалізовані такі основні зв’язки:

– кожен користувач може мати декілька звернень у ролі клієнта, а також може бути призначеним спеціалістом для багатьох звернень;

– кожне звернення може містити посилання на прикріплений звіт та пов’язане з багатьма повідомленнями;
– повідомлення пов’язані із зверненням, містять автора (користувача), а також можуть містити прикріплений звіт;
– кожен звіт створюється певним користувачем і може бути пов’язаний із зверненням або повідомленням.

Кардинальність зв’язків забезпечує цілісність даних та відповідає основним сценаріям використання системи. Така структура дозволяє ефективно виконувати запити, зберігати історію спілкування та роботи із зверненнями, а також забезпечує можливість подальшого розширення функціоналу системи.

2.4 Проектування модулів системи

2.4.1 Модуль реєстрації та автентифікації користувачів

Модуль реєстрації та автентифікації користувачів у програмному комплексі призначений для забезпечення створення, підтвердження та ідентифікації облікових записів користувачів системи. Його головна мета — гарантувати безпечний доступ до функціоналу комплексу тільки для авторизованих осіб, відповідно до їхньої ролі.

Архітектура модуля передбачає окремі процеси для реєстрації нового користувача (клієнта) і автентифікації (логіну) для всіх ролей — адміністратора, спеціаліста та клієнта.

Реєстрація користувача (клієнта) ініціюється на клієнтській частині (JavaFX додаток), де користувач вводить ім’я, електронну пошту та пароль. Після введення даних клієнт надсилає їх на сервер за допомогою HTTP-запиту до ендпоінта /api/auth/sign-up. Сервер здійснює перевірку валідності отриманої інформації, а також контролює унікальність вказаної електронної адреси. У разі успішної валідації дані тимчасово зберігаються у кеші (Caffeine Cache) до моменту

підтвердження e-mail. Після цього користувач має можливість ініціювати надсилання коду підтвердження на свою електронну пошту. Введення отриманого коду у відповідне поле в застосунку дозволяє завершити процес валідації. Якщо код підтвердження введено коректно, сервер створює нового користувача в базі даних із роллю “клієнт” і повертає клієнтському додатку JWT-токен разом з інформацією про користувача для подальшої роботи із системою.

Послідовність взаємодії основних компонентів під час реєстрації користувача, надсилання та підтвердження коду, а також отримання JWT-токена зображено на рисунку 2.3.

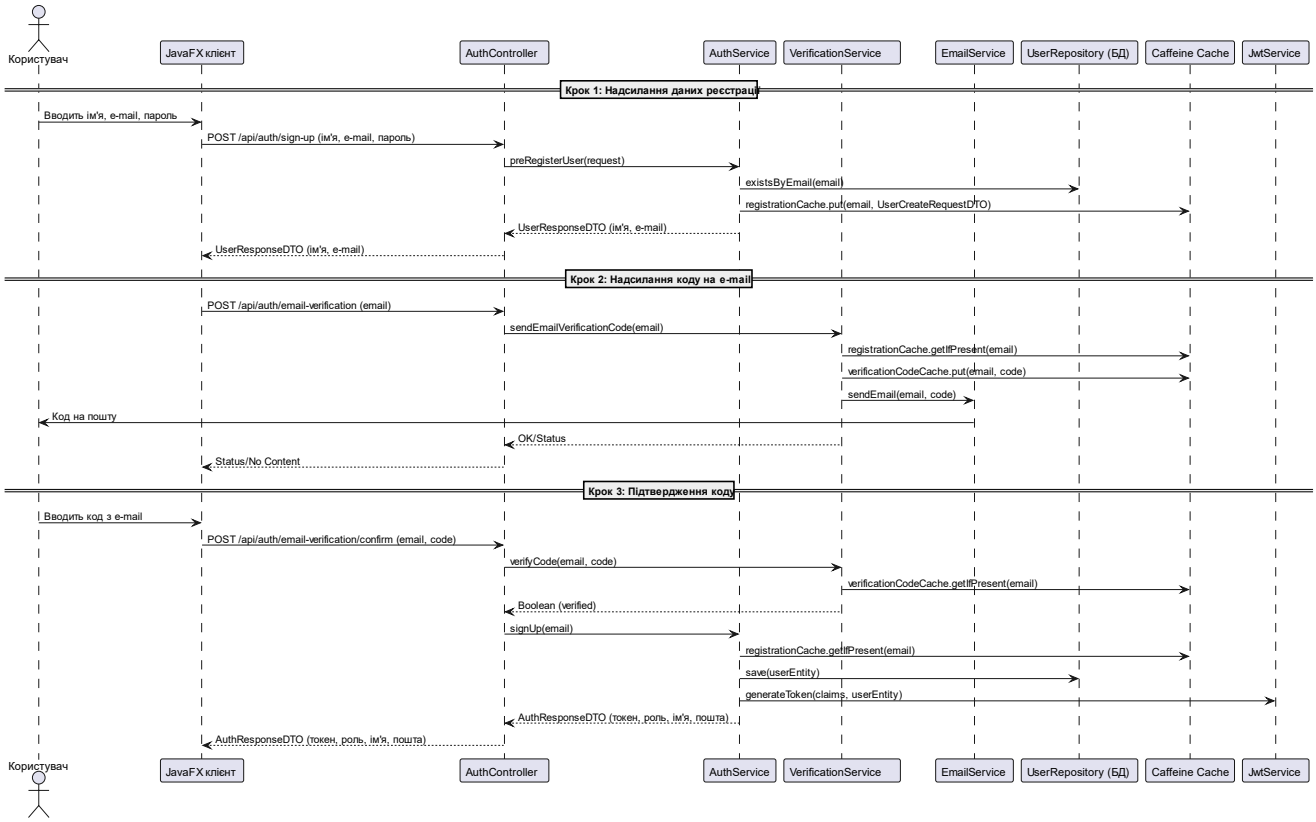


Рисунок 2.3 – Діаграма послідовності сценарію реєстрації користувача з підтвердженням e-mail

Автентифікація користувачів (логін) реалізована однаково для всіх ролей. Користувач вводить e-mail та пароль, після чого ці дані надсилаються на сервер через HTTP-запит до /api/auth/sign-in. Сервер перевіряє облікові дані за допомогою

сервісу автентифікації. У разі успіху клієнту повертається JWT-токен із інформацією про роль користувача. Токен надалі використовується для доступу до захищених ендпоінтів API.

Взаємодія компонентів модуля організована наступним чином:

– authController приймає всі запити, пов’язані з реєстрацією, логіном, надсиланням та підтвердженням коду;

– authService обробляє основну логіку реєстрації, автентифікації та створення користувача;

– verificationService відповідає за генерацію, зберігання та перевірку коду підтвердження e-mail;

– emailService відправляє електронні листи з кодом на e-mail користувача;

– userRepository взаємодіє з базою даних для збереження та отримання користувачів;

– caffeine cache використовується для тимчасового зберігання даних реєстрації та кодів підтвердження;

– jwtService генерує та перевіряє JWT-токени для автентифікації.

Основні сценарії використання модуля:

– реєстрація нового клієнта — відбувається тільки для ролі “клієнт”; спеціалісти та адміністратори створюються адміністратором або під час ініціалізації системи;

– вхід до системи — можливий для всіх ролей, з перевіркою логіна та паролю, незалежно від статусу ролі.

Безпека та захист:

– паролі користувачів зберігаються у базі лише у захешованому вигляді;

– для підтвердження реєстрації використовується одноразовий код, що надсилається на e-mail;

– для всіх захищених запитів клієнт повинен передавати дійсний JWT-токен у заголовку Authorization;

2.4.2 Модуль збору системних параметрів

Модуль збору системних параметрів реалізовано на клієнтській частині системи як окремий сервіс, що інтегрується у JavaFX-застосунок і використовує бібліотеку OSHI-core. Проєктування модуля передбачає послідовну роботу сервісу. Спочатку користувач у графічному інтерфейсі ініціює збір системних параметрів, після чого CrossPlatformDataGatherService отримує необхідні характеристики обладнання та операційної системи, сформована структура даних пакує зібрану інформацію для передавання, а завершальним кроком ці дані надсилаються на сервер як REST-запит для подальшого формування звіту.

Зібраний модулем набір параметрів включає:

– характеристики операційної системи (назва, версія, виробник, архітектура, аптайм, час завантаження, кількість процесів і потоків, права користувача);

– детальну інформацію про центральний процесор (модель, виробник, тактова частота, кількість ядер, завантаження тощо);

– обсяг, тип і швидкість оперативної пам’яті та swap, деталі фізичних і віртуальних модулів;

– параметри графічних адаптерів (назва, виробник, драйвер, обсяг відеопам’яті);

– дані про накопичувачі (модель, обсяг, розділи, файлові системи, активність, вільне/зайняте місце);

– інформацію про батарею, живлення, заряд, виробника, температуру, напругу, статус зарядки;

– показники температури, напруги, обертів вентиляторів (системні датчики);

– характеристики мережевих інтерфейсів (назва, тип, статус, швидкість, MAC/IP-адреси, трафік, DNS, шлюзи, статистика TCP/UDP);

– дані про звукові, USB-пристрої, підключені монітори (EDID, Hex);

– відомості про найресурсомісткіші запущені процеси (PID, шлях, споживання ОЗП та ЦП, кількість потоків, стан).

Усі ці параметри упаковуються у структуру та асинхронно надсилаються серверу через REST API для формування діагностичного звіту. Модуль обробляє можливі помилки збору та інформує користувача про статус операції у графічному інтерфейсі.

Завдяки цьому модулю у сервісному центрі отримують максимально повну і структуровану картину стану обладнання, що значно підвищує якість підтримки та пришвидшує вирішення технічних питань.

2.4.3 Модуль формування звітів

Модуль формування звітів реалізований на серверній частині системи і відповідає за створення структурованого PDF-документа з отриманих системних параметрів. Модуль приймає на вхід структуру даних, яка містить зібрані клієнтом апаратні та програмні характеристики комп’ютера користувача, і формує на їх основі звіт у форматі PDF.

Генерація PDF здійснюється із використанням бібліотеки Apache PDFBox. При створенні звіту модуль структурує дані по розділах, відповідно до їхньої природи (операційна система, комп’ютерна система, процесор, пам’ять, накопичувачі, мережа, батарея, активні процеси тощо). Форматування та наповнення кожної секції здійснюється згідно із заданою специфікацією, що забезпечує зручність аналізу для спеціаліста сервісного центру.

Після генерації PDF-файлу модуль зберігає його у базі даних як масив байтів, разом із супутньою інформацією (користувач, дата, ім’я файлу). У відповідь на запит клієнта модуль повертає унікальний ідентифікатор звіту для подальшої ідентифікації в системі.

Робота модуля відбувається у кілька етапів. Спочатку він приймає структуровані дані від клієнтської частини через REST API, потім формує PDF-документ за заданою структурою, далі зберігає його у базі даних як BLOB (Binary Large Object) разом із метаданими і, на завершення, повертає клієнтові ідентифікатор створеного звіту.

Приклад повністю сформованого PDF-звіту наведено у Додатку В.

2.4.4 Модуль комунікації з сервісним центром

Модуль комунікації з сервісним центром відповідає за організацію інтерактивного обміну повідомленнями між клієнтом та спеціалістом для кожного окремого звернення (issue). Створення звернення здійснюється клієнтом через HTTP-запит, у якому вказуються тема, опис і, за необхідності, ідентифікатор раніше сформованого звіту. Після цього система автоматично створює новий чат із унікальним ідентифікатором.

Всі подальші повідомлення в межах звернення обробляються у реальному часі за допомогою протоколу WebSocket із використанням STOMP. Для кожного чату створюється окремий топік (/topic/chat/{issueId}), на який підписуються учасники (клієнт і спеціаліст). Повідомлення надсилаються через WebSocket і одразу стають доступними обом сторонам, а також зберігаються у базі даних.

Модуль підтримує наступні основні функції:

– ініціація звернення через REST-запит, із можливістю прикріпити звіт;

– обмін повідомленнями між клієнтом і спеціалістом у реальному часі через WebSocket;

– прикріплення звіту до повідомлення чату;

– зміна статусу чату (OPEN, IN_PROGRESS, CLOSED) залежно від дій користувачів;

– можливість закриття чату як клієнтом, так і спеціалістом, із автоматичним інформуванням через WebSocket про зміну статусу;

– контроль доступу (обмінюватися повідомленнями можуть лише учасники відповідного звернення, що перевіряється на основі авторизації).

Всі операції чату супроводжуються логуванням подій та обробкою можливих помилок. Модуль базується на архітектурі Spring WebSocket (STOMP) та інтегрується із загальною системою контролю доступу (Spring Security).

2.5 Проектування інтерфейсу користувача

Інтерфейс користувача програмного комплексу реалізовано на базі JavaFX із використанням FXML-файлів для опису вигляду вікон та елементів керування. Дизайн інтерфейсу розроблено з урахуванням принципів зручності, інтуїтивності й уніфікованості для всіх типів користувачів.

До спільних елементів інтерфейсу для всіх ролей належать:

– єдина форма входу до системи (див. рисунок Б.1);

– стандартне вікно редагування профілю (див. рисунок Б.4);

– повідомлення про статуси, помилки та успішні дії, що відображаються у вигляді підписів;

– підтримка української мови.

Окрім цього, інтерфейс чату для клієнта та спеціаліста також є ідентичним.

Індивідуальні особливості інтерфейсу кожної ролі переважно стосуються центральної частини вікна, де розміщується основний функціонал.

Для клієнта, спеціаліста та адміністратора є детальна покрокова інструкція роботи .

2.5.1 Інтерфейс клієнта та спеціаліста

Спільна частина для клієнта та спеціаліста:

– лівий блок для списку чатів зі статусом;

– вікно чату, яке містить історію повідомлень, поле для введення тексту, кнопка “відправити”, кнопка “закрити чат”.

Приклад такого інтерфейсу наведено на рисунку Б.5.

Особливості для клієнта:

– екран реєстрації нового користувача (див. рисунок Б.2);

– вікно підтвердження e-mail при реєстрації (див. рисунок Б.3);

– центральна частина вікна “Кабінет користувача” містить кнопку “Згенерувати звіт”, форму для створення звернення (див. рисунок Б.6);

– права панель з кнопкою “Мої звіти”, яка відкриває вікно для перегляду, завантаження і видалення звітів (див. рисунок Б.7).

Особливість інтерфейсу спеціаліста є відображення у центральній частині вікна статистики — кількість «доступних» та «призначених» звернень, що показано на рис. Б.8.

2.5.2 Інтерфейс адміністративної частини

Інтерфейс адміністратора має такі особливості:

– ліва панель містить список усіх спеціалістів із можливістю видалення;

– у центральній частині розташовано форму для створення нового спеціаліста.

Приклад такого інтерфейсу наведено на рисунку Б.9.

2.6 Висновки до розділу 2

У другому розділі виконано повний цикл проектування програмного комплексу моніторингу системних параметрів і обліку клієнтських звернень для сервісного центру.

Були визначені функціональні та нефункціональні вимоги до системи, що охоплюють підтримку різних ролей користувачів, безпеку даних, масштабованість, продуктивність і зручність інтерфейсу. Запропоновано архітектуру на основі клієнт-серверної моделі, що забезпечує чітке розмежування відповідальності між компонентами комплексу та сприяє його подальшому розвитку.

Описано структуру бази даних, основні сутності та їх взаємозв’язки, що дозволяють забезпечити цілісність даних і підтримати всі функціональні сценарії системи. Детально розглянуто проектування ключових модулів — реєстрації та автентифікації користувачів, збору системних параметрів, формування звітів і організації комунікації між клієнтами та спеціалістами.

Особливу увагу приділено проектуванню інтерфейсу користувача для всіх ролей із урахуванням принципів зручності, інтуїтивності та модульності. Наведено діаграми, що ілюструють архітектуру компонентів системи.

Таким чином, у розділі закладено технічні основи для подальшої реалізації, тестування й впровадження програмного комплексу.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ

3.1 Розробка серверної частини

Серверна частина програмного комплексу реалізована на базі фреймворку Spring Boot. Вона забезпечує обробку запитів від клієнтської частини через REST API, реалізує основну бізнес-логіку, а також взаємодіє з базою даних для зберігання інформації про користувачів, звернення, повідомлення та звіти.

3.1.1 Реалізація REST API для взаємодії з клієнтською частиною

Для організації взаємодії клієнтської частини із сервером створено низку REST API контролерів, що відповідають за виконання ключових операцій системи.

AuthController — керує операціями автентифікації та реєстрації користувачів, включно з логіном, реєстрацією, надсиланням і підтвердженням коду з електронної пошти, а також генерацією JWT-токенів для безпечного доступу до системи.

ClientController — реалізує функціонал створення та перегляду звернень (чатів), отримання історії повідомлень у чатах, формування та отримання системних звітів користувача, а також керування станом звернень.

SpecialistController — надає спеціалістам сервісного центру можливість отримувати список своїх звернень, переглядати історію чатів, отримувати статистику, завантажувати прикріплені звіти та закривати звернення.

AdminController — забезпечує адміністративне управління системою, зокрема додавання, перегляд і видалення спеціалістів сервісного центру.

Вхідні та вихідні дані REST API представлені у вигляді структурованих об'єктів — DTO (Data Transfer Object), що забезпечують чітку структуру та типізацію інформації, обмін якою відбувається у форматі JSON.

Для захисту API реалізовано механізм аутентифікації та авторизації на основі JWT-токенів, які передаються у заголовках HTTP-запитів. Сервер перевіряє валідність токена перед обробкою кожного запиту, що забезпечує безпеку доступу до ресурсів системи.

Обробка помилок виконується централізовано через глобальний обробник виключень, який формує відповіді у форматі Problem Details (RFC 7807). Це дозволяє клієнтам отримувати структуровану і зрозумілу інформацію про помилки, що виникають під час роботи з API.

Реалізація контролерів побудована на декларативних анотаціях Spring, таких як @RestController, @RequestMapping, @PostMapping, @GetMapping тощо. Вхідні дані валідовані за допомогою анотації @Valid на рівні DTO.

Вихідний код REST-контролерів, наведено у Додатку А.

3.1.2 Реалізація бізнес-логіки сервісного центру

Основна бізнес-логіка сервісного центру зосереджена у сервісних класах, які обробляють ключові процеси системи — управління користувачами, зверненнями, чат-повідомленнями та формування звітів. Кожен сервіс інкапсулює власну предметну область, що спрощує супровід і розширення функціональності.

AuthService відповідає за автентифікацію й реєстрацію користувачів. Він:

– шифрує паролі;

– кешує тимчасові дані (DTO реєстрації та коди верифікації) у Caffeine Cache;

– використовує AuthenticationManager для перевірки облікових даних.

Алгоритм обробки запиту входу починається зі зчитування електронної адреси й пароля з тіла запиту, далі ці дані передаються в AuthenticationManager, який виконує автентифікацію. Якщо перевірка неуспішна, сервіс генерує BadCredentialsException з HTTP-кодом 401. Коли автентифікація пройдена, із SecurityContextHolder береться об’єкт UserEntity, формуються claims із ключем

role, після чого jwtService.generateToken створює JWT-токен. Далі UserEntity перетворюється на UserResponseDTO, і клієнту повертається AuthResponseDTO, що містить токен, дані профілю та ідентифікатор користувача.

Послідовність реєстрації з підтвердженням електронної пошти демонструється на діаграмі 2.3.

AdminService відповідає за адміністрування сервісного центру — створення, перегляд та видалення спеціалістів.

Алгоритм додавання спеціаліста спершу перевіряє, чи не зайнята вказана електронна пошта. Якщо адреса вільна, на основі отриманих даних створюється UserEntity з роллю SPECIALIST, об’єкт зберігається у базі, після чого клієнту повертаються дані нового спеціаліста.

Для отримання списку формується пагінований запит до бази даних і повертається результат у вигляді Page<UserResponseDTO>.

При видаленні перевіряється наявність запису з указаним ID, якщо такий існує, запис видаляється, інакше генерується виняток із кодом HTTP 409.

ClientService реалізує логіку роботи клієнта: створює й керує зверненнями, веде історію чатів, генерує та зберігає PDF-звіти й надає їх користувачеві. Послідовність дій під час формування звіту показано на рис. 3.1, а приклад готового PDF наведено у Додатку В.

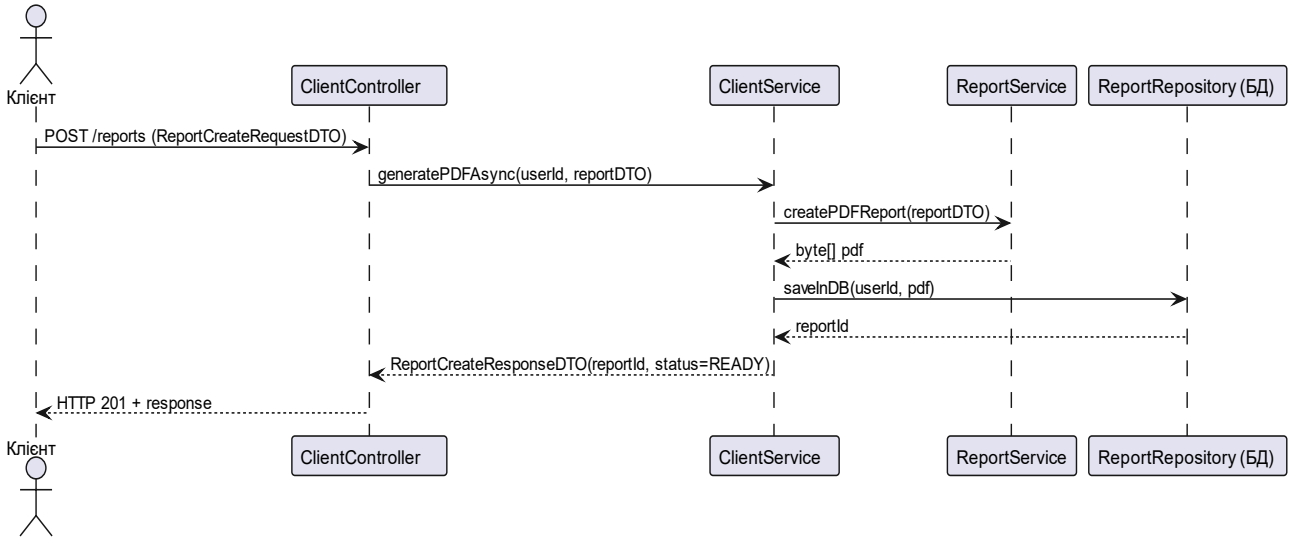


Рисунок 3.1 – Діаграма послідовності створення системного звіту

Алгоритм створення звернення передбачає послідовні дії сервісу ClientService. Спершу у базі даних визначається обліковий запис активного користувача; якщо до звернення додається системний звіт, сервіс перевіряє, чи такий файл уже збережено. Далі формується об’єкт IssueEntity, що містить дані клієнта, опис проблеми й статус OPEN, і цей об’єкт одразу фіксується у базі. Після цього зберігається перше повідомлення чату з текстом опису. На завершення клієнту повертається відповідь DTO із даними про нове звернення.

Для отримання списку звернень формується пагінований запит до бази даних і повертається результат.

Для отримання історії повідомлень у конкретному зверненні перевіряється, чи належить це звернення клієнту; у разі успіху повертається пагінований список повідомлень.

Для закриття звернення спочатку перевіряється, чи є користувач власником цього звернення, якщо так, статус звернення оновлюється на CLOSED, і про зміну статусу інформуються всі підписники відповідного чату через WebSocket.

Для роботи зі звітами (перегляд, видалення, створення PDF) також здійснюється перевірка прав доступу. Звіти, пов’язані з користувачем, повертаються у вигляді пагінованого списку, а при спробі видалення або отримання PDF перевіряється, чи належить звіт поточному користувачу, інакше генерується виняток.

SpecialistService забезпечує функціонал для спеціалістів сервісного центру — отримання списку звернень, перегляд статистики, історії чатів, завантаження прикріплених звітів, а також закриття звернень.

Для отримання списку звернень формується пагінований запит до бази даних і результаті повертається список для поточного спеціаліста.

Для отримання статистики звернень викликається агрегований запит до репозиторію за ID спеціаліста.

Для перегляду повідомлень у зверненні перевіряється чи це звернення вже призначене спеціалісту, якщо ні — призначається автоматично й оновлюється

статус звернення на "IN_PROGRESS". Якщо перевірка успішна, повертається пагінований список повідомлень.

Для закриття звернення перевіряється чи є поточний користувач відповідальним спеціалістом. Якщо так — статус звернення оновлюється на CLOSED і всі підписники відповідного чату інформуються про зміну статусу через WebSocket.

Для отримання PDF-звіту спочатку перевіряється, чи належить відповідне звернення поточному спеціалісту та чи був звіт дійсно прикріплений до цього звернення; у разі невідповідності генерується виняток.

UserService відповідає за оновлення профілів користувачів із підтвердженням змін через код, що надсилається на електронну пошту. Для цього використовується кешування тимчасових даних оновлення та верифікаційного коду. Алгоритм оновлення профілю складається з ініціації та підтвердження змін.

На етапі ініціації сервіс перевіряє, чи не використовується нова електронна пошта іншим користувачем, а також знаходить поточного користувача у базі даних. Якщо користувач вказав новий пароль, він одразу шифрується. Після цього формується унікальний ідентифікатор (UUID), під яким кешуються нові дані профілю. На цю ж адресу електронної пошти (нову або стару) надсилається випадковий код для підтвердження змін, який також кешується у системі.

Під час підтвердження змін користувач передає UUID і код. Сервіс перевіряє, чи існує активна сесія оновлення і чи збігається введений код із тим, що був надісланий на електронну пошту. У разі успішної перевірки оновлюються відповідні поля профілю користувача у базі даних, а кеш оновлення та код верифікації видаляються. Клієнту повертається оновлена інформація про користувача.

Усі сервіси виконують валідацію вхідних даних, обробку виключень та логують ключові події для подальшого аналізу та підтримки.

Для обробки повідомлень чатів використовується ChatService, який зберігає повідомлення, перевіряє права користувачів на звернення, а також розсилає повідомлення через WebSocket брокер.

Код сервісних класів наведено у Додатку А.

3.1.3 Реалізація вебсокетів для обміну повідомленнями у чаті

Обмін повідомленнями у режимі реального часу між клієнтами та спеціалістами сервісного центру реалізований за допомогою протоколу WebSocket із використанням STOMP (Simple Text Oriented Messaging Protocol). Це дозволяє забезпечити асинхронну, двонаправлену комунікацію, що гарантує миттєве надсилання та отримання чат-повідомлень без необхідності постійно оновлювати сторінку або надсилати повторні HTTP-запити.

Конфігурація WebSocket-сервера, реалізована у класі WebSocketConfig, визначає кінцеву точку /ws, через яку клієнти встановлюють з'єднання. Для маршрутизації повідомлень використовується брокер повідомлень, налаштований із префіксами /queue, /topic та /user. Ці префікси дозволяють реалізувати приватні та групові канали обміну повідомленнями, а також підтримують адресацію повідомлень конкретним користувачам.

Безпека WebSocket-з'єднань забезпечується спеціальним конфігураційним класом WebSocketSecurityConfig. Цей клас реалізує авторизацію на рівні повідомлень, контролюючи доступ користувачів до конкретних тем (topic) та особистих каналів (user) згідно з їх ролями та правами доступу. Авторизація здійснюється на основі JWT-токенів, які клієнт передає при встановленні сесії.

При ініціалізації роботи з чатом клієнт спочатку виконує стандартний WebSocket handshake через HTTP-запит на адресу /ws, що дозволяє оновити протокол зв’язку. Після встановлення базового WebSocket-з’єднання клієнт ініціює сесію STOMP, надсилаючи STOMP CONNECT повідомлення з токеном автентифікації. Сервер, у свою чергу, виконує перевірку токена через механізми відповіддю STOMP CONNECTED. Цей процес зображено на рисунку 3.2.

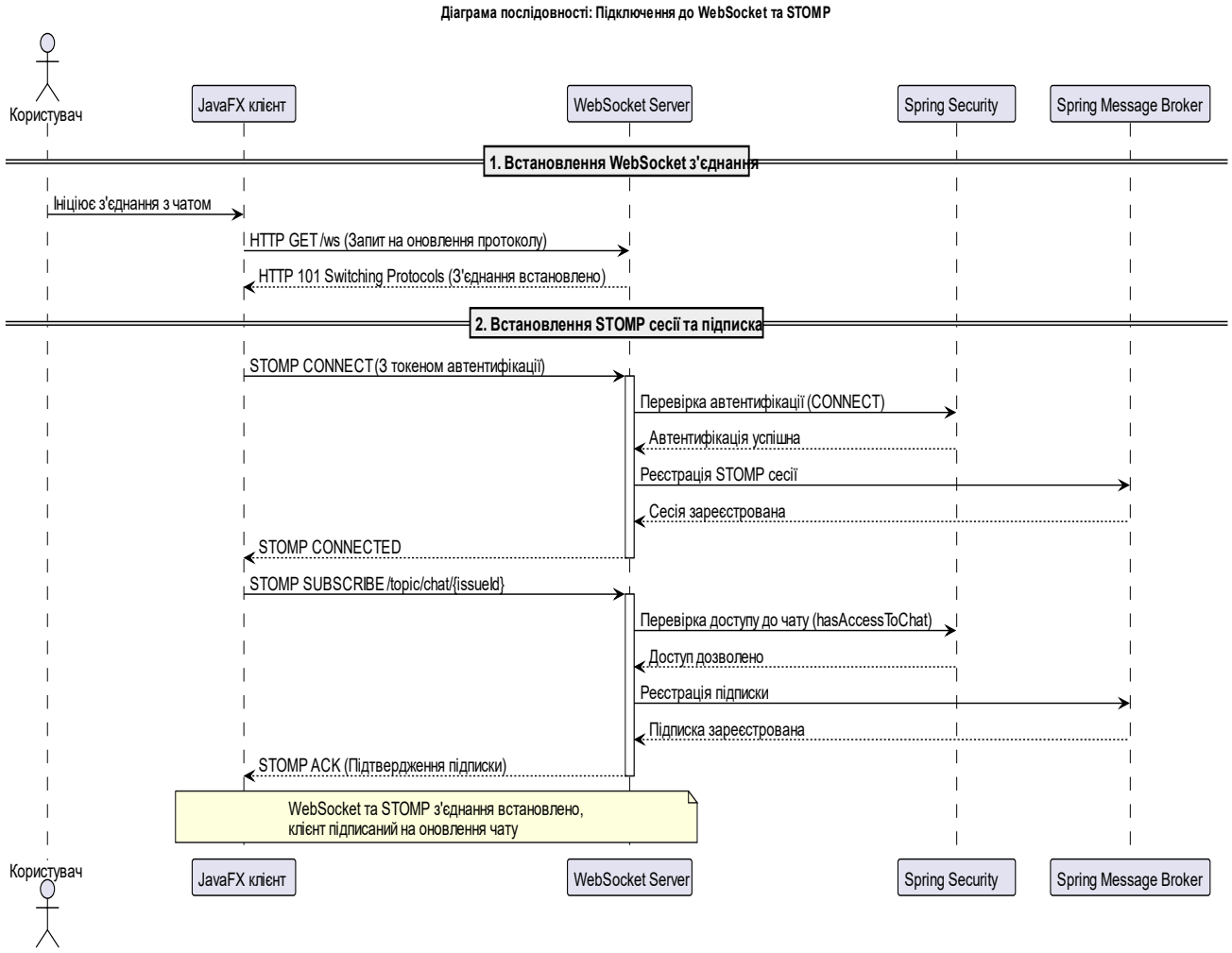


Рисунок 3.2 – Послідовність підключення і підписки WebSocket

Після встановлення сесії клієнт підписується на необхідні канали:

– /topic/chat/{issueId} — канал обміну повідомленнями у конкретному чаті;

– /topic/chat-status/{issueId} — канал оновлення статусів чатів (наприклад, закриття звернення).

Обробка вхідних повідомлень здійснюється у контролері ChatController, який приймає повідомлення за адресою /app/chat/{issueId}/send. Кожне повідомлення проходить валідацію та зберігається у базі даних через сервіс ChatService, де також перевіряється, чи має користувач право надсилати повідомлення у цей чат (чи він є клієнтом чи спеціалістом, пов’язаним із відповідним зверненням).

Після збереження повідомлення воно миттєво транслюється всім підписникам каналу через брокер Spring, що гарантує оперативне оновлення чат-інтерфейсу на стороні користувачів.

Для контролю доступу користувачів до повідомлень та каналів у чаті використовується механізм, який перевіряє, чи пов’язаний конкретний користувач із відповідним зверненням (Issue). Це забезпечує конфіденційність даних і запобігає несанкціонованому доступу.

3.2 Розробка клієнтської частини

3.2.1 Реалізація модуля моніторингу з використанням OSHI-core

Для реалізації модуля моніторингу системних параметрів використано Java-бібліотеку OSHI-core, яка забезпечує кросплатформений доступ до апаратної та системної інформації. Основна логіка збору даних реалізована у сервісі CrossPlatformDataGatherService, який інтегрований у клієнтську частину програмного комплексу.

Сервіс CrossPlatformDataGatherService збирає широкий спектр інформації, що включає параметри процесора, оперативної пам’яті, накопичувачів, відеокарт, датчиків, мережевих інтерфейсів, операційної системи, а також інформацію про активні процеси з високим навантаженням. Зібрані дані формуються у структуру ReportCreateRequestDTO, що передається на сервер для подальшої обробки та формування звітів.

Ініціація збору даних та взаємодія з користувачем здійснюються через контролер ClientCenterViewController. Він відповідає за запуск процесу збору системної інформації асинхронно (для уникнення блокування інтерфейсу), відправлення зібраних даних на сервер, а також за отримання статусу обробки звіту та можливість його завантаження у форматі PDF.

Ключові методи контролера включають обробку натискання кнопки "Згенерувати звіт", виконання збору системних параметрів у окремому потоці, обробку відповіді сервера та оновлення інтерфейсу користувача згідно зі статусом звіту.

Передача даних між клієнтською частиною і сервером здійснюється через REST API, використовуючи структуру ReportCreateRequestDTO. Для завантаження згенерованих звітів застосовується спеціалізований сервіс SystemReportService, який відповідає за формування HTTP-запитів та обробку відповідей.

Детальний код реалізації сервісу CrossPlatformDataGatherService та контролера ClientCenterViewController наведено у Додатку А лістинги А.12 та А.13 відповідно.

3.2.2 Реалізація користувацького інтерфейсу

Користувацький інтерфейс програмного комплексу розроблено з використанням JavaFX з описом вигляду у FXML-файлах, що забезпечує гнучкість у побудові компонентів і зручність підтримки. Взаємодія користувача з додатком реалізована через контролери Java, які обробляють події, керують станом елементів інтерфейсу та взаємодіють із сервісним шаром.

Інтерфейс підтримує три основні ролі користувачів — клієнт, спеціаліст та адміністратор, при цьому для всіх ролей реалізовано уніфіковані базові елементи:

– форма входу;
– редагування профілю (поточні дані користувача);
– відображення повідомлень про статус дій;

Основний віконний простір динамічно завантажує компоненти залежно від ролі користувача, що дозволяє відокремити функціональні області для кожної ролі. Для клієнта передбачено кабінет користувача з можливістю генерації системних звітів, створення звернень, перегляду статусу та завантаження звітів. Спеціаліст має інтерфейс для моніторингузвернень, перегляду чатів та статистики. Адміністратор керує списком спеціалістів та їх даними.

Реалізація інтерфейсу побудована на принципах асинхронності — важливі операції, такі як генерація звітів, виконуються у фонових потоках, а оновлення

інтерфейсу здійснюється за допомогою Platform.runLater(), що гарантує безпеку роботи з JavaFX UI-потоком.

Для оформлення застосовано набір CSS-стилів, які підтримують консистентність дизайну, адаптивність і зручність користування.

Ключовий контролер інтерфейсу клієнта — ClientCenterViewController, який відповідає за генерацію та завантаження звітів, створення звернень і відображення статусів операцій. Для спеціаліста та адміністратора також розроблені власні контролери (наприклад, SpecialistCenterViewController (див. А.14) та AdminCenterViewController), які реалізують специфічний функціонал для відповідних ролей — моніторинг і обробку звернень, роботу зі статистикою та керування спеціалістами. Логіка обробки подій і взаємодії з інтерфейсом у цих контролерах загалом є уніфікованою.

3.3.1 Захист персональних даних користувачів

Захист персональних даних користувачів у розробленому програмному комплексі здійснюється за допомогою комплексу заходів, спрямованих на забезпечення конфіденційності, цілісності та доступності інформації.

Для безпечного збереження паролів користувачів застосовується хешування з використанням BCrypt – сучасного та стійкого алгоритму, який унеможливлює відновлення оригінального пароля з хешу навіть у разі компрометації бази даних.

Зміни персональних даних (ім'я, електронна пошта, пароль) супроводжуються механізмом верифікації через одноразовий код, що надсилається на електронну пошту користувача. Такий підхід запобігає несанкціонованому оновленню інформації і забезпечує додатковий рівень контролю.

Обмін повідомленнями у чатах, який відбувається через WebSocket, захищений на рівні протоколу Spring Security WebSocket. Використовується система авторизації повідомлень, яка перевіряє, чи має користувач право отримувати інформацію із певного каналу (topic). Реалізовано механізм контролю доступу, який дозволяє підписатися лише на ті чат-канали, що пов’язані із зверненнями користувача, запобігаючи витоку інформації (див. лістинг А.16).

Таким чином, поєднання криптографічних методів, механізмів аутентифікації та авторизації, а також верифікації через електронну пошту забезпечує надійний захист персональних даних користувачів від несанкціонованого доступу та викрадення.

3.3.2 Захист комунікації між клієнтом та сервером

Захист комунікації між клієнтом та сервером у програмному комплексі реалізовано за допомогою сучасних механізмів аутентифікації та авторизації на основі JWT-токенів (JSON Web Token). Кожен користувач після успішної автентифікації отримує унікальний токен, який надалі передається у заголовках усіх запитів до серверу — як для REST API, так і для WebSocket-з'єднань.

На серверній стороні застосовується Spring Security у поєднанні зі спеціалізованим JWT-фільтром (JwtAuthenticationFilter), який перехоплює всі вхідні HTTP-запити, перевіряє наявність та валідність токена і витягує інформацію про користувача для подальшої авторизації (див. А.17). Такий підхід дозволяє централізовано контролювати доступ до приватних ресурсів та реалізовувати гнучкі політики безпеки.

Контроль ролей користувача та обмеження доступу до функціоналу здійснюються згідно з принципами мінімальних прав тобто сервер перевіряє валідність токена, визначає роль користувача та приймає рішення про доступ до відповідного ресурсу. У разі невалідного або відсутнього токена користувач отримує відповідь із відмовою у доступі.

Для WebSocket-комунікації механізм перевірки прав реалізований на рівні підключення та підписки до каналів. При встановленні з’єднання і підписці на повідомлення токен також перевіряється через Spring Security, і лише авторизовані користувачі можуть отримувати повідомлення, що стосуються їхніх звернень або чатів.

Така архітектура забезпечує цілісність та конфіденційність даних, а також виключає можливість несанкціонованого доступу до функціональних можливостей комплексу.

Конфігурація розмежування прав доступу, керування аутентифікацією та авторизацією, а також підключення JWT-фільтра реалізована у класі SecurityConfiguration. У цьому класі визначено дозволи для різних кінцевих точок системи відповідно до ролей користувачів, а також здійснюється реєстрація JWT-фільтра, який перевіряє токен при кожному запиті (див. А.18).

3.4 Інтеграція компонентів системи

Інтеграція компонентів програмного комплексу забезпечує безперебійну взаємодію між серверною та клієнтською частинами, а також цілісність і узгодженість даних у процесі роботи системи.

Серверна частина, розроблена на Spring Boot, взаємодіє з реляційною базою даних за допомогою Hibernate (JPA), що забезпечує об’єктно-реляційне відображення (ORM) даних. Завдяки цьому реалізується прозоре збереження, оновлення та вилучення сутностей без необхідності прямої роботи з SQL-запитами. Використання ORM сприяє підвищенню продуктивності розробки і зниженню ймовірності помилок у роботі з базою.

Клієнтська частина, створена на JavaFX, взаємодіє із сервером через REST API для отримання та передачі даних, а також через WebSocket для реалізації функціоналу чату в режимі реального часу. Модуль моніторингу на стороні клієнта використовує бібліотеку OSHI-core для збору системної інформації, яка надалі передається на сервер для формування звітів.

Комунікація між клієнтом і сервером відбувається через стандартизовані протоколи HTTP(S) та WebSocket, що гарантує надійність та ефективність передачі даних.

Загальна архітектура системи передбачає чітке розмежування обов’язків між компонентами та використання сучасних технологій, що забезпечує гнучкість,

масштабованість і простоту підтримки. Діаграма компонентів детально описана у розділі 2.2.

3.5 Висновки до розділу 3

У третьому розділі виконано безпосередню реалізацію програмного комплексу згідно із спроєктованою архітектурою, функціональними і нефункціональними вимогами.

Описано основні підходи до реалізації серверної частини на основі Spring Boot із чітким поділом на REST API, сервісні класи для бізнес-логіки, а також механізми обробки помилок, аутентифікації та авторизації. Запроваджено сучасні засоби захисту даних, включаючи хешування паролів, валідацію даних і централізовану обробку виключень.

Особливу увагу приділено реалізації модуля чату на WebSocket/STOMP, що дозволило організувати асинхронний обмін повідомленнями між клієнтом і спеціалістом у реальному часі із забезпеченням контролю доступу.

Розглянуто реалізацію клієнтської частини — десктопного застосунку JavaFX, що містить модуль збору системної інформації (на базі OSHI-core), динамічний інтерфейс та інструменти для асинхронної роботи з сервером і відображення статусу операцій.

Забезпечено надійний захист персональних даних через механізми автентифікації/авторизації на основі JWT, а також організовано захищений обмін даними між клієнтом і сервером (у тому числі для WebSocket-з’єднань).

Розкрито питання інтеграції компонентів системи — як на рівні технологічної взаємодії (HTTP, WebSocket, ORM), так і на рівні організації коду та логічного поділу обов’язків.

Таким чином, у цьому розділі реалізовано всі основні технічні рішення, що дозволили отримати повнофункціональний і захищений програмний комплекс, готовий до тестування

4 ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ ПРОГРАМНОГО КОМПЛЕКСУ

4.1 Методологія тестування

Для перевірки працездатності, функціональності та стабільності програмного комплексу була застосована комплексна методологія тестування, що поєднує ручні та автоматизовані підходи.

Основним методом тестування обрано ручне функціональне тестування ключових сценаріїв користувача. Перевірка виконувалась для всіх основних ролей (клієнт, спеціаліст, адміністратор) шляхом імітації реальних дій у клієнтському застосунку. Перевірка виконувалась для таких основних сценаріїв:

– реєстрація користувача;
– автентифікація користувача;
– генерація системних звітів;
– створення звернень;
– обробка звернень;
– робота з чатом;
– адміністративні операції.

Додатково було використано автоматизоване тестування окремих модулів на рівні модульних тестів (JUnit) для перевірки коректності бізнес-логіки серверної частини, а також інтеграційне тестування REST API за допомогою Postman та Swagger UI.

Для оцінки стабільності роботи системи під навантаженням було проведено навантажувальне тестування з використанням Gatling [4]. Цей підхід дозволив

перевірити реакцію системи на одночасні запити від великої кількості користувачів, зокрема операції авторизації та генерації звітів.

Усі критичні шляхи, помилки та результати тестування фіксувалися в окремій таблиці, що дозволило відстежувати виконання вимог та оперативно усувати недоліки.

4.2 Функціональне тестування системи

Функціональне тестування проводилося з метою перевірки реалізації усіх заявлених функціональних вимог програмного комплексу для різних ролей користувачів. Тестування здійснювалося вручну шляхом проходження основних користувацьких сценаріїв у клієнтському застосунку та перевірки відповідної реакції серверної частини.

Для кожної функціональної вимоги було розроблено окремий тестовий сценарій, що охоплює весь життєвий цикл відповідної функції — від ініціації дії користувачем до очікуваного результату в системі. Оцінка виконання тестів проводилась за фактом досягнення очікуваного результату.

Результати функціонального тестування системи узагальнено у таблиці 4.1. У ній наведено перелік основних функціональних вимог для всіх ролей, відповідні сценарії тестування, очікувані результати та фактичний стан виконання.

Таблиця 4.1 – Результати функціонального тестування функціональних вимог системи

Функціональна вимога	Роль	Сценарій тестування	Очікуваний результат	Факт виконання
Перегляд і редагування профілю з підтвердженням коду на e-mail	Всі	Зміна імені/пошти/паролю, підтвердження коду	Дані змінено тільки після підтвердження коду	+

Продовження таблиці 4.1

Автоматичне створення облікового запису адміністратора при першому запуску	Адмін	Перший запуск серверної частини	Можливість увійти як адміністратор	+
Перегляд списку спеціалістів	Адмін	Перехід до списку спеціалістів	Відображається список	+
Додавання нового спеціаліста	Адмін	Введення ім’я, e-mail, пароля, підтвердження	Спеціаліст додається до списку	+
Видалення спеціаліста	Адмін	Видалення спеціаліста зі списку	Спеціаліст видаляється	+
Вхід до системи за e-mail і паролем (аккаунт створює адміністратор)	Спеціаліст	Вхід під створеним адміністратором аккаунтом	Вхід успішний	+
Перегляд кількості відкритих звернень (Open/In Progress)	Спеціаліст	Перевірка відображення статистики	Коректна статистика відображається	+
Доступ до списку всіх звернень, де призначений спеціаліст	Спеціаліст	Перегляд звернень	Всі доступні звернення відображаються	+
Відкриття чату звернення, перегляд/надсилання повідомлень, завантаження звіту	Спеціаліст	Відкриття чату, надсилання повідомлень	Повідомлення відправляються і приймаються, звіт завантажується	+
Призначення себе відповідальним при відкритті Open-звернення	Спеціаліст	Відкриття звернення зі статусом "Open"	Звернення переходить у статус "In Progress"	+
Закриття звернення	Спеціаліст	Переведення звернення у статус "Closed"	Звернення закривається, чат блокується	+
Реєстрація з підтвердженням e-mail	Клієнт	Реєстрація нового користувача, підтвердження коду	Аккаунт активується тільки після підтвердження	+
Вхід до системи	Клієнт	Вхід з вірними/невірними даними	Доступ/відмова	+
Генерація звіту	Клієнт	Клік на кнопку "Згенерувати звіт"	Звіт створюється, зберігається у системі	+

Кінець таблиці 4.1

Завантаження згенерованого звіту	Клієнт	Завантаження PDF-звіту	PDF звіт завантажується	+
Перегляд та видалення власних звітів	Клієнт	Видалення власного звіту	Звіт видаляється із системи	+
Створення нового звернення до сервісного центру	Клієнт	Заповнення форми звернення, прикріплення звіту	Звернення створюється, з’являється у списку	+
Перегляд списку чатів звернень за статусами	Клієнт	Відкриття списку звернень	Відображаються Open, In Progress, Closed	+
Відкриття чату звернення, обмін повідомленнями зі спеціалістом	Клієнт	Надсилання та отримання повідомлень	Повідомлення доставляються та відображаються	+
Закриття звернення (зміна статусу на "Closed")	Клієнт	Переведення звернення у "Closed"	Звернення закривається, чат стає недоступним	+

4.3 Тестування зручності використання (Usability Testing)

Тестування зручності використання програмного комплексу проводилося методом експертної оцінки шляхом самостійного проходження основних сценаріїв користування для всіх ролей системи. Перевірялися такі аспекти, як інтуїтивність інтерфейсу, доступність основних функцій, наявність повідомлень про помилки, а також логічність навігації між розділами.

За результатами тестування встановлено, що основні функціональні можливості реалізовані у вигляді окремих модулів, доступ до яких здійснюється з головного меню згідно з роллю користувача. Операції створення звернень, генерації звітів, обміну повідомленнями в чаті та адміністративного керування спеціалістами виконуються просто та логічно.

Виявлено, що для операції видалення звітів або спеціалістів наразі відсутній механізм додаткового підтвердження дії, що може бути джерелом випадкового видалення даних. Рекомендується у наступних версіях реалізувати підтвердження для критичних операцій.

Загалом інтерфейс оцінено як зручний, зрозумілий та такий, що відповідає вимогам до сучасних десктопних застосунків.

4.4 Навантажувальне тестування

Для перевірки стійкості та продуктивності серверної частини програмного комплексу було проведено навантажувальне тестування за допомогою інструменту Gatling, який дозволяє моделювати одночасну роботу багатьох користувачів, аналізувати часи відповіді серверу, пропускну здатність і надійність виконання транзакцій.

Сценарій тестування передбачав емулювання одночасної роботи 50 унікальних користувачів, для кожного з яких виконувався ланцюжок – вхід до системи та масова генерація звітів.

При вході до системи кожен користувач здійснює запит на автентифікацію через REST-ендпоінт /api/auth/sign-in із унікальною парою e-mail/пароль, отримуючи у відповідь JWT-токен.

Масова генерація звітів відбувалася для всіх успішно аутентифікованих користувачів. Кожен користувач здійснює 20 послідовних запитів на створення системного звіту (/api/clients/me/reports), використовуючи один і той самий тестовий JSON-звіт у тілі запиту та отриманий токен у заголовку.

Між логіном і першою генерацією звіту витримувалася пауза у 10 секунд (імітація переходу користувача до наступного кроку).

Gatling-тест реалізовано у вигляді окремого проєкту (див. лістинг А.19 у Додатку А). Для моделювання сценарію використано csv-файл із 50 унікальними користувачами.

Під час тесту було виконано 1050 запитів, 50 логінів та 1000 створень звітів, усі вони виконалися успішно. Оцінка швидкодії та пропускної здатності здійснювалася за наступними метриками на рисунку 4.1.

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	1050	1050	-	0	26.92	131	330	397	5122	10852	11911	840	2006
User Login	50	50	-	0	1.28	6464	9330	10578	11694	11911	11911	9162	1856
Create Report 0	50	50	-	0	1.28	166	2474	4839	5110	5197	5197	2628	1731
Create Report 1	50	50	-	0	1.28	334	470	514	582	622	622	488	65
Create Report 2	50	50	-	0	1.28	184	406	447	496	606	606	376	107
Create Report 3	50	50	-	0	1.28	145	427	448	470	486	486	370	112
Create Report 4	50	50	-	0	1.28	162	400	428	468	518	518	360	98
Create Report 5	50	50	-	0	1.28	151	381	438	499	513	513	343	117
Create Report 6	50	50	-	0	1.28	150	356	375	419	488	488	329	74
Create Report 7	50	50	-	0	1.28	144	342	365	392	420	420	298	68
Create Report 8	50	50	-	0	1.28	137	300	332	372	399	399	266	63
Create Report 9	50	50	-	0	1.28	141	307	335	377	398	398	270	64
Create Report 10	50	50	-	0	1.28	137	326	344	380	666	666	283	103
Create Report 11	50	50	-	0	1.28	137	299	328	347	360	360	260	78
Create Report 12	50	50	-	0	1.28	137	307	336	367	387	387	268	84
Create Report 13	50	50	-	0	1.28	137	323	347	369	383	383	283	79
Create Report 14	50	50	-	0	1.28	131	322	339	367	479	479	294	77
Create Report 15	50	50	-	0	1.28	134	306	330	365	377	377	275	79
Create Report 16	50	50	-	0	1.28	136	313	328	373	387	387	282	80
Create Report 17	50	50	-	0	1.28	139	312	326	385	465	465	287	77
Create Report 18	50	50	-	0	1.28	140	318	331	361	404	404	284	73
Create Report 19	50	50	-	0	1.28	142	234	338	356	361	361	245	86

Рисунок 4.1 – Статистика виконання запитів у Gatling

Результати для сценарію автентифікації користувача показують, що середній час відповіді становив 9162 мс, медіанне значення часу відповіді — 9330 мс. Максимальний зафіксований час відповіді під час виконання запиту на вхід до системи сягав 11911 мс.

У процесі масової генерації звітів для кожного користувача було виконано 20 послідовних запитів на створення звіту. Середній час відповіді для цих операцій варіювався від 245 мс до 2628 мс, найвищий показник спостерігався для першого запиту у кожній серії. Медіанні значення часу відповіді на запити створення звітів перебували у діапазоні від 234 мс до 2474 мс, що також підтверджує тенденцію до збільшення тривалості виконання першої операції у циклі.

4.5 Аналіз результатів тестування

Усі основні функціональні вимоги, заявлені у специфікації системи, були реалізовані та підтверджені у процесі ручного тестування.

За результатами тестування зручності використання встановлено, що інтерфейс є зрозумілим і доступним для користувача, основні операції виконуються інтуїтивно, а повідомлення про статуси дій коректно відображаються для всіх ролей. Разом із тим було ідентифіковано відсутність додаткового підтвердження для критичних операцій видалення, що може підвищити ризик випадкової втрати даних. Це слід врахувати під час подальшого розвитку системи.

Аналіз результатів навантажувального тестування показав, що при моделюванні одночасного входу 50 користувачів (login) сервер на базі інстансу AWS t2.micro справився з усіма запитами без збоїв. Однак середній час відповіді для операції входу склав близько 9 секунд, а максимальний — майже 12 секунд, що є очікуваним для ресурсів цього класу. Операції масової генерації звітів виконувалися значно швидше. Після першого (найповільнішого) запиту подальші створення звітів для всіх користувачів оброблялись із середнім часом відповіді у межах 250–500

Таким чином, результати тестування підтверджують загальну працездатність і відповідність програмного комплексу встановленим вимогам. Система демонструє стійкість до паралельних навантажень, коректну обробку основних бізнес-процесів і зручність використання для кінцевих користувачів.

4.6 Рекомендації щодо впровадження та підтримки системи

Для успішного впровадження та подальшої стабільної роботи програмного комплексу важливо забезпечити належну підготовку інфраструктури. Розгортання

серверної частини бажано здійснювати на сучасному обладнанні з достатнім обсягом оперативної пам’яті та процесорних ресурсів, що дозволить підтримувати продуктивність системи при роботі під навантаженням. У виробничому середовищі доцільно використовувати сервери з кращими характеристиками, ніж t2.micro, оскільки це забезпечить вищу надійність та швидкодію.

Подальша експлуатація програмного комплексу має супроводжуватися регулярним резервним копіюванням бази даних, що дозволить мінімізувати ризики втрати інформації у разі виникнення збоїв чи аварійних ситуацій. Систематичний моніторинг логів і поточного стану сервера сприятиме своєчасному виявленню та усуненню потенційних проблем.

Підтримка та оновлення програмного комплексу повинні здійснюватися із урахуванням його модульної архітектури, що дозволяє розширювати функціонал без критичних змін основної логіки. Регулярне оновлення програмних компонентів рекомендується виконувати з метою підвищення захищеності системи від можливих уразливостей і забезпечення надійної роботи.

4.7 Висновки до розділу 4

Проведений цикл тестування програмного комплексу засвідчив відповідність розробленої системи усім заявленим функціональним і експлуатаційним вимогам. За результатами ручного тестування для ролей адміністратора, спеціаліста й клієнта було підтверджено реалізацію повного спектру користувацьких сценаріїв, а жодних критичних дефектів, що перешкоджають роботі, не виявлено. Експертна оцінка зручності використання засвідчила інтуїтивність інтерфейсу та логічне розміщення основних функцій. Разом з тим, було ідентифіковано відсутність механізму підтвердження для критичних операцій видалення, що може підвищити ризик випадкової втрати даних; це рекомендовано врахувати у подальших ітераціях розвитку комплексу.

Результати навантажувального тестування на інстансі AWS t2.micro засвідчили стійкість системи при одночасній роботі 50 користувачів. Усі запити були оброблені без втрат, а середній час відповіді при логіні склав близько 9 секунд, що є очікуваним для подібної інфраструктури. Операції масової генерації звітів виконувалися істотно швидше — після першого запиту час відповіді становив у середньому 250–500 мс. Таким чином, отримані показники свідчать про готовність системи до експлуатації у виробничому середовищі, хоча для підвищення продуктивності доцільно масштабувати серверну інфраструктуру.

Дослідження підтвердило, що програмний комплекс має сучасну модульну архітектуру, зручний для користувача інтерфейс, а також супроводжується інструментами для резервного копіювання та журналювання. Рекомендовано надалі впровадити підтвердження для операцій видалення критичних даних, налаштувати моніторинг продуктивності та резервне копіювання, а також розглянути можливість перенесення серверу на потужніший хост для скорочення часу автентифікації.

Загалом проведене тестування засвідчило високу якість, надійність і зручність експлуатації системи, а також окреслило можливі напрями подальшого вдосконалення й оптимізації програмного комплексу для забезпечення його ефективної роботи в довгостроковій перспективі.

ВИСНОВКИ

У ході виконання дипломної кваліфікаційної роботи було розроблено та впроваджено програмний комплекс моніторингу системних параметрів і обліку клієнтських звернень для сервісного центру. В роботі виконано повний цикл від аналізу предметної області та сучасних інструментів до розробки, тестування й рекомендацій щодо впровадження готової системи.

Проведено детальний аналіз сучасних підходів до збору системної інформації комп'ютера, охарактеризовано основні бібліотеки та утиліти, зокрема обґрунтовано вибір бібліотеки OSHI-core для кросплатформеного збору даних на Java. Здійснено огляд існуючих рішень і сформульовано функціональні й нефункціональні вимоги до майбутньої системи, що дозволило закласти фундамент для якісного проєктування.

У процесі проектування було побудовано архітектуру програмного комплексу на основі клієнт-серверної моделі, де серверна частина реалізована із використанням Spring Boot, а клієнтська — на JavaFX. Розроблено структуру бази даних та ключові модулі системи, що забезпечують надійний збір, зберігання й обробку даних, а також підтримують захист персональних даних, зручність використання і масштабованість.

Реалізовано серверну частину з REST API, підтримкою WebSocket/STOMP для онлайн-чату та генерацією PDF-звітів, а також клієнтську частину з інтерактивним інтерфейсом для різних ролей користувачів. Особливу увагу приділено питанням безпеки — реалізовано механізми автентифікації й авторизації на основі JWT, захищено передачу даних та впроваджено багаторівневий контроль доступу.

Виконане тестування, включаючи функціональне, навантажувальне та оцінку зручності інтерфейсу, засвідчило відповідність програмного комплексу встановленим вимогам. Система показала стійкість до паралельних навантажень, коректну обробку бізнес-процесів і високу зручність для користувача. За

результатами тестів сформульовано практичні рекомендації щодо оптимізації серверної інфраструктури та подальшого розвитку системи.

Таким чином, розроблений програмний комплекс успішно вирішує поставлені завдання. Він забезпечує ефективний збір системних параметрів, автоматизацію обробки клієнтських звернень, інтерактивну взаємодію між клієнтом і сервісним центром, а також генерацію звітів у зручному форматі. Структура системи, модульність і використання сучасних технологій дозволяють легко масштабувати й адаптувати комплекс під потреби підприємства, а також впроваджувати нові функції в майбутньому.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. AIDA64 Documentation. Офіційна документація до AIDA64. URL: https://www.aida64.com/user-manual. (дата звернення: 01.05.2025).

2. Apache PDFBox Documentation. Офіційна документація до Apache PDFBox для генерації PDF-документів. URL: https://pdfbox.apache.org. (дата звернення: 16.04.2025).

3. CPU-Z. Утиліта для отримання інформації про процесор, материнську плату та оперативну пам’ять. URL: https://www.cpuid.com/softwares/cpu-z.html. (дата звернення: 01.05.2025).

4. Gatling Documentation. Офіційна документація до інструмента навантажувального тестування Gatling. URL: https://docs.gatling.io (дата звернення: 29.05.2025).

5. Hibernate ORM Documentation. Офіційна документація Hibernate ORM для об’єктно-реляційного відображення. URL: https://hibernate.org/orm/documentation. (дата звернення: 30.04.2025).

6. HWiNFO Software. Офіційний сайт інструменту HWiNFO. URL: https://www.hwinfo.com. (дата звернення: 01.05.2025).

7. Java Native Access (JNA) Documentation. Документація для використання JNA, який є основою для OSHI. URL: https://github.com/java-native-access/jna. (дата звернення: 30.04.2025).

8. JavaFX Documentation. Офіційна документація JavaFX для створення графічних інтерфейсів. URL: https://openjfx.io/. (дата звернення: 24.04.2025).

9. JWT Introduction. Офіційний вступ до JSON Web Token — відкритого стандарту для безпечного обміну інформацією між сторонами. URL: https://jwt.io/introduction. (дата звернення: 15.05.2025).

10. OSHI Documentation. Офіційна документація до бібліотеки OSHI для збору інформації про ПК. URL: https://github.com/oshi/oshi. (дата звернення: 26.04.2025).

11. PostgreSQL Documentation. Офіційна документація до PostgreSQL для роботи з базами даних. URL: https://www.postgresql.org/docs/. (дата звернення: 30.04.2025).

12. Speccy. Офіційна сторінка утиліти від Piriform. URL: https://www.ccleaner.com/speccy. (дата звернення: 01.05.2025).

13. Spring Boot Documentation. Офіційна документація до Spring Boot для створення серверної частини. URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle. (дата звернення: 29.04.2025).

14. Spring Documentation. Офіційна документація Spring Framework для ін’єкції залежностей та створення бекенд-додатків. URL: https://docs.spring.io/spring-framework/reference/index.html. (дата звернення: 25.04.2025).

15. Spring Security Documentation. Офіційна документація до Spring Security для реалізації механізмів аутентифікації, авторизації та захисту вебзастосунків. URL: https://docs.spring.io/spring-security/reference/. (дата звернення: 15.05.2025).

16. Spring WebSocket Documentation. Документація до Spring WebSocket для організації чату в реальному часі. URL: https://docs.spring.io/spring-framework/reference/web/websocket.html. (дата звернення: 30.04.2025).

17. WMIC (Windows Management Instrumentation Command-line). Документація для використання WMIC на Windows для збору системної інформації. URL: https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page. (дата звернення: 22.04.2025).

ДОДАТОК А
ВИХІДНИЙ КОД КЛЮЧОВИХ КОМПОНЕНТІВ СИСТЕМИ

Лістинг А.1 – Клас AuthController, що реалізує операції аутентифікації та реєстрації користувачів

```
@RestController
@RequestMapping("/api/auth")
@Slf4j
@RequiredArgsConstructor
@Tag(name = "Authentication", description = "User registration and authorization operations")
public class AuthController {

    private final AuthService authService;
    private final VerificationService verificationService;

    @PostMapping("/sign-in")
    public ResponseEntity<AuthResponseDTO> signIn(@Valid @RequestBody SignInRequestDTO request) {
        log.info("Controller was invoked by sign-in request");
        AuthResponseDTO authResponse = authService.signIn(request);

        return ResponseEntity.ok(authResponse);
    }

    @PostMapping("/sign-up")
    public ResponseEntity<UserResponseDTO> signUp(@Valid @RequestBody UserCreateRequestDTO request) {
        log.info("Controller was invoked by sign-up request");

        UserResponseDTO userResponseDTO = authService.preRegisterUser(request);
```

```
        return ResponseEntity.ok(userResponseDTO);
    }

    @PostMapping("/email-verification")
    public ResponseEntity<Void> sendEmailVerificationCode(@Valid
@RequestBody SendEmailCodeRequestDTO sendEmailRequest) {
        log.info("Controller was invoked by email verification request
to send code");

verificationService.sendEmailVerificationCode(sendEmailRequest.email
());
        return ResponseEntity.noContent().build();
    }

    @PostMapping("/email-verification/confirm")
    public ResponseEntity<AuthResponseDTO>
verifyEmailVerificationCode(
            @Valid @RequestBody ConfirmEmailCodeRequestDTO
confirmCodeRequest) {
        log.info("Controller was invoked by email verification request
to verify code");
        String email = confirmCodeRequest.email();
        boolean verified = verificationService.verifyCode(email,
confirmCodeRequest.code());

        if (!verified) {
            throw new InvalidVerificationCodeException("Invalid
verification code");
        }
        AuthResponseDTO authResponseDTO = authService.signUp(email);
        return
ResponseEntity.status(HttpStatus.CREATED).body(authResponseDTO);
    }
}
```

Лістинг А.2 – Клас AdminController, що реалізує адміністративне керування спеціалістами

```
@Controller
@RequestMapping("/api/admin/app")
@RequiredArgsConstructor
@Slf4j
public class AdminController {
    private final AdminService adminService;

    @PostMapping("/specialists")
    public ResponseEntity<UserResponseDTO> addSpecialist(@Valid
@RequestBody UserCreateRequestDTO userCreateRequestDTO) {
        UserResponseDTO userResponseDTO =
adminService.addSpecialist(userCreateRequestDTO);

        return ResponseEntity.ok(userResponseDTO);
    }

    @GetMapping("/specialists")
    public ResponseEntity<SpecialistListResponseDTO> getSpecialists(
            @PositiveOrZero @RequestParam(defaultValue = "0") int
page,
            @Positive @RequestParam(defaultValue = "10") int size) {
        log.info("Getting specialists from page {} and size: {}",
page, size);
        PageRequest pageRequest = PageRequest.of(page, size,
Sort.by("nickname").ascending());

        Page<UserResponseDTO> specialistPage =
adminService.getSpecialists(pageRequest);

        SpecialistListResponseDTO specialistListResponseDTO =
SpecialistListResponseDTO.builder()
                .currentPage(specialistPage.getNumber())
                .totalElements(specialistPage.getTotalElements())
```

```
                .totalPages(specialistPage.getTotalPages())
                .specialists(specialistPage.getContent())
                .build();

        return ResponseEntity.ok(specialistListResponseDTO);
    }

    @DeleteMapping("/specialists/{specialistId}")
    public ResponseEntity<Void> deleteSpecialist(@Positive
@PathVariable Integer specialistId) {
        adminService.deleteSpecialist(specialistId);

        return ResponseEntity.noContent().build();
    }
}
```

Лістинг А.3 – Клас ClientController, що обробляє операції зі зверненнями та звітами клієнта

```
@RestController
@RequestMapping("/api/clients/me")
@Slf4j
@RequiredArgsConstructor
public class ClientController {
    private final ClientService clientService;

    @PostMapping("/issues")
    public ResponseEntity<IssueCreateResponseDTO> createIssue(
            @Valid @RequestBody IssueCreateRequestDTO
issueCreateRequestDTO) {
        log.info("Create issue request: {}", issueCreateRequestDTO);
        IssueCreateResponseDTO response =
clientService.createIssue(issueCreateRequestDTO);
```

```
        return
ResponseEntity.status(HttpStatus.CREATED).body(response);
    }

    @GetMapping("/issues")
    public                          ResponseEntity<IssueListResponseDTO>
getIssues(@PositiveOrZero @RequestParam int page,
                                                                @Positive
@RequestParam int size) {
        log.debug("Getting issues from page " + page + " and size " +
size);
        PageRequest pageRequest = PageRequest.of(page, size);
        Page<IssueGetResponseDTO>            issuesPage            =
clientService.getClientIssues(pageRequest);

        IssueListResponseDTO response = new IssueListResponseDTO(
                issuesPage.getContent(),
                issuesPage.getTotalPages(),
                issuesPage.getTotalElements(),
                issuesPage.getNumber()
        );
        return ResponseEntity.ok(response);
    }

    @GetMapping("/issues/{issueId}/messages")
    public                      ResponseEntity<MessageListResponseDTO>
getIssueMessages(@Positive @PathVariable Integer issueId,

@PositiveOrZero @RequestParam int page,

@Positive @RequestParam int size) {
        log.info("Getting messages from page {} and size {} for issue:
{}", page, size, issueId);
        PageRequest   pageRequest   =   PageRequest.of(page,   size,
Sort.by("sentAt").ascending());
```

```
        Page<MessageSendResponseDTO> issuesPage = clientService.getIssueMessages(issueId, pageRequest);

        MessageListResponseDTO response = new MessageListResponseDTO(
                issuesPage.getContent(),
                issuesPage.getTotalPages(),
                issuesPage.getTotalElements(),
                issuesPage.getNumber()
        );

        log.info("MessageListResponseDTO return: {}", response.getMessages());
        return ResponseEntity.ok(response);
    }

    @PatchMapping("/issues/{issueId}")
    public ResponseEntity<Void> closeIssue(@PathVariable Integer issueId) {
        log.info("Closing issue {}", issueId);
        clientService.closeIssue(issueId);

        return ResponseEntity.noContent().build();
    }

    @PostMapping("/reports")
    public ResponseEntity<ReportCreateResponseDTO> createPDFReport(
            @Valid @AuthenticationPrincipal UserDetails userDetails,
            @Valid @RequestBody ReportCreateRequestDTO reportCreateRequestDTO) {
        String userId = userDetails.getUsername();
        log.info("createPDFReport was invoked by user with id: {}", userId);
```

```
        ReportCreateResponseDTO reportCreateResponseDTO =
clientService.generatePDFAsync(userId, reportCreateRequestDTO);

        return
ResponseEntity.status(HttpStatus.CREATED).body(reportCreateResponseD
TO);
    }

    @GetMapping("/reports")
    public ResponseEntity<ReportListResponseDTO>
getClientReports(@PositiveOrZero @RequestParam int page,

@Positive @RequestParam int size) {
        log.trace("Getting reports from page {} and size {}", page,
size);
        PageRequest pageRequest = PageRequest.of(page, size,
Sort.by("createdAt").descending());

        Page<ReportGetResponseDTO> reportsPage =
clientService.getClientReports(pageRequest);

        ReportListResponseDTO response = new ReportListResponseDTO(
                reportsPage.getContent(),
                reportsPage.getTotalPages(),
                reportsPage.getTotalElements(),
                reportsPage.getNumber()
        );

        return ResponseEntity.ok(response);
    }

    @DeleteMapping("/reports/{reportId}")
    public ResponseEntity<Void> deleteClientReport(@Positive
@PathVariable Integer reportId) {
        clientService.deleteClientReportById(reportId);
```

```
        return ResponseEntity.noContent().build();
    }

    @GetMapping("/reports/{reportId}")
    @Operation(summary = "Get PDF report by ID", description =
"Retrieves a generated PDF report as a file")
    public ResponseEntity<byte[]> getPDFReportById(@Positive
@PathVariable Integer reportId) {
        ReportEntity reportEntity =
clientService.getPDFReport(reportId);
        byte[] pdfBytes = reportEntity.getReport();

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_PDF);
        String filename = reportEntity.getFilename();

headers.setContentDisposition(ContentDisposition.builder("attachment
")
                .filename(filename)
                .build());

        headers.setContentLength(pdfBytes.length);

        return ResponseEntity.ok().headers(headers).body(pdfBytes);
    }
}
```

Лістинг А.4 – Клас SpecialistController, що надає функції для роботи спеціалістів із зверненнями

```
@RestController
@RequestMapping("/api/specialists/me")
@Slf4j
public class SpecialistController {
    private final SpecialistService specialistService;
```

```
    public SpecialistController(SpecialistService specialistService)
{
        this.specialistService = specialistService;
    }

    @GetMapping("/issues")
    public                        ResponseEntity<IssueListResponseDTO>
getIssues(@PositiveOrZero @RequestParam int page,
                                                          @Positive
@RequestParam int size) {
        log.info("Getting issues from page " + page + " and size " +
size);
        PageRequest pageRequest = PageRequest.of(page, size);

        Page<IssueGetResponseDTO>            issuesPage            =
specialistService.getSpecialistIssues(pageRequest);

        IssueListResponseDTO response = new IssueListResponseDTO(
                issuesPage.getContent(),
                issuesPage.getTotalPages(),
                issuesPage.getTotalElements(),
                issuesPage.getNumber()
        );

        return ResponseEntity.ok(response);
    }

    @GetMapping("/issues/{issueId}/messages")
    public                     ResponseEntity<MessageListResponseDTO>
getIssueMessages(@Positive @PathVariable Integer issueId,

@PositiveOrZero          @RequestParam          int          page,
@Positive @RequestParam int size) {
        log.info("Getting messages from page {} and size {} for issue:
{}", page, size, issueId);
```

```
        PageRequest pageRequest = PageRequest.of(page, size,
Sort.by("sentAt").ascending());

        Page<MessageSendResponseDTO> issuesPage =
specialistService.getIssueMessages(issueId, pageRequest);

        MessageListResponseDTO response = new MessageListResponseDTO(
                issuesPage.getContent(),
                issuesPage.getTotalPages(),
                issuesPage.getTotalElements(),
                issuesPage.getNumber()
        );

        return ResponseEntity.ok(response);
    }

    @GetMapping("/issues/stats")
    public ResponseEntity<IssueStatProjection> getIssueStats() {
        log.info("Getting issues stats");
        IssueStatProjection issueStatsResponseDTO =
specialistService.getIssueStats();
        return ResponseEntity.ok(issueStatsResponseDTO);
    }

    @PatchMapping("/issues/{issueId}")
    public ResponseEntity<Void> closeIssue(@Positive @PathVariable
Integer issueId) {
        log.info("Closing issue {}", issueId);
        specialistService.closeIssue(issueId);

        return ResponseEntity.noContent().build();
    }

    @GetMapping("/issues/{issueId}/reports/{reportId}")
```

```
    public ResponseEntity<byte[]> getPDFReportById(@Positive
@PathVariable Integer issueId,
                                                   @Positive
@PathVariable Integer reportId) {
        ReportEntity reportEntity =
specialistService.getPDFReport(issueId, reportId);
        byte[] pdfBytes = reportEntity.getReport();

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_PDF);
        String filename = reportEntity.getFilename();

headers.setContentDisposition(ContentDisposition.builder("attachment
")
                .filename(filename)
                .build());

        headers.setContentLength(pdfBytes.length);

        return ResponseEntity.ok().headers(headers).body(pdfBytes);
    }
}
```

Лістинг А.5 – Клас UserController, що відповідає за оновлення та підтвердження змін профілю користувача

```
@RestController
@RequestMapping("/api/users/me")
@RequiredArgsConstructor
@Slf4j
public class UserController {
    private final UserService userService;

    @PutMapping("/profile")
```

```
    public ResponseEntity<ProfilePutResponseDTO>
initiateUpdateProfile(@Valid @RequestBody ProfilePutRequestDTO
profilePutRequestDTO) {
        log.info("ProfilePutRequestDTO: {}", profilePutRequestDTO);
        ProfilePutResponseDTO profilePutResponseDTO =
userService.initiateProfileUpdate(profilePutRequestDTO);

        return ResponseEntity.ok(profilePutResponseDTO);
    }

    @PutMapping("/profile/confirm")
    public ResponseEntity<UserResponseDTO>
updateProfileConfirm(@Valid @RequestBody ProfileConfirmDTO
profilePutConfirmDTO) {
        log.info("ProfileConfirmDTO: {}", profilePutConfirmDTO);
        UserResponseDTO userResponseDTO =
userService.confirmProfileUpdate(profilePutConfirmDTO);

        return ResponseEntity.ok(userResponseDTO);
    }
}
```

Лістинг А.6 – Клас AuthService на сервері — логіка автентифікації та реєстрації користувачів

```
@Service
@Slf4j
@RequiredArgsConstructor
public class AuthService {
    private final JwtService jwtService;
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authManager;
    private final UserMapper userMapper;
```

```
    private final Cache<String, UserCreateRequestDTO>
registrationCache;

    public AuthResponseDTO signUp(String email) {
        log.info("Sign up for user: {}", email);
        UserCreateRequestDTO userCreateRequestDTO =
registrationCache.getIfPresent(email);

        UserEntity userEntity =
userMapper.signUpRequestDTOToUserEntity(userCreateRequestDTO);

        userEntity.setRole(Role.CLIENT);

        userEntity = userRepository.save(userEntity);

        log.info("Successful sign up attempt for user: {}",
userEntity.getEmail());
        Map<String, Object> claims = new HashMap<>();
        claims.put("role", userEntity.getRole());

        String jwt = jwtService.generateToken(claims, userEntity);

        UserResponseDTO userResponseDTO =
userMapper.userEntityToUserResponseDTO(userEntity);

        return new AuthResponseDTO(jwt, userResponseDTO,
userEntity.getId());
    }

    public AuthResponseDTO signIn(SignInRequestDTO request) {
        log.info("Sign in for user: {}", request.email());

        Authentication authentication = authManager.authenticate(
```

```
                new
UsernamePasswordAuthenticationToken(request.email(),
request.password())
        );

SecurityContextHolder.getContext().setAuthentication(authentication;

        log.info("Try get userEntity from security context");
        UserEntity         userEntity         =          (UserEntity)
authentication.getPrincipal();

        log.info("Successful   login   attempt   for   user:   {}",
request.email());

        Map<String, Object> claims = new HashMap<>();
        claims.put("role", userEntity.getRole());

        String jwt = jwtService.generateToken(claims, userEntity);

        UserResponseDTO                userResponseDTO                =
userMapper.userEntityToUserResponseDTO(userEntity);

        return     new     AuthResponseDTO(jwt,     userResponseDTO,
userEntity.getId());
    }

    public    UserResponseDTO    preRegisterUser(UserCreateRequestDTO
request) {
        if (userRepository.existsByEmail(request.email())) {
            throw   new   ResponseStatusException(HttpStatus.CONFLICT,
"User with such e-mail is already exist");
        }

        UserCreateRequestDTO       requestWithEncodedPassword       =
UserCreateRequestDTO.builder()
```

```
                .email(request.email())

.password(passwordEncoder.encode(request.password()))
                .nickname(request.nickname())
                .build();

        registrationCache.put(request.email(),
requestWithEncodedPassword);

        UserResponseDTO                userResponseDTO                =
userMapper.userCreateRequestDTOToUserResponseDTO(requestWithEncodedP
assword);
        userResponseDTO.setRole(Role.CLIENT);
        return userResponseDTO;
    }
}
```

Лістинг А.7 – Клас AdminService на сервері — адміністрування спеціалістів сервісного центру

```
@Service
@RequiredArgsConstructor
@Slf4j
public class AdminService {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final UserMapper userMapper;

    public     UserResponseDTO     addSpecialist(UserCreateRequestDTO
userCreateRequestDTO) {
        if
(userRepository.existsByEmail(userCreateRequestDTO.email())) {
            throw  new  MyResponseStatusException(HttpStatus.CONFLICT,
"Email already exists");
        }
```

```
        UserEntity                    userEntity                    =
userMapper.signUpRequestDTOToUserEntity(userCreateRequestDTO);
        userEntity.setRole(Role.SPECIALIST);

userEntity.setPassword(passwordEncoder.encode(userCreateRequestDTO.p
assword()));

        userEntity = userRepository.save(userEntity);

        return userMapper.userEntityToUserResponseDTO(userEntity);
    }

    public Page<UserResponseDTO> getSpecialists(Pageable pageRequest)
{
        return          userRepository.findAllByRole(Role.SPECIALIST,
pageRequest)
                .map(userMapper::userEntityToUserResponseDTO);
    }

    public void deleteSpecialist(Integer specialistId) {
        if (!userRepository.existsById(specialistId)) {
            throw new MyResponseStatusException(HttpStatus.CONFLICT,
"Specialist does not exist");
        }
        userRepository.deleteById(specialistId);
    }
}
```

Лістинг А.8 – Клас ClientService на сервері — управління клієнтськими зверненнями та звітами

```
@Slf4j
@RequiredArgsConstructor
@Service
public class ClientService {
```

```
    private final UserRepository userRepository;
    private final ReportRepository reportRepository;
    private final IssueRepository issueRepository;
    private final MessageRepository messageRepository;
    private final ReportService reportService;
    private final IssueMapper issueMapper;
    private final MessageMapper messageMapper;
    private final ReportMapper reportMapper;
    private final SimpMessagingTemplate broker;

    public IssueCreateResponseDTO createIssue(IssueCreateRequestDTO
createRequest) {
        log.debug("Creating new issue {}", createRequest);

        Integer userId = getCurrentUserId();

        UserEntity userEntity = userRepository.findById(userId)
                .orElseThrow(()                ->                new
ResourceNotFoundException("User not found"));

        ReportEntity reportEntity = null;
        Integer reportId = createRequest.reportId();
        if (reportId != null) {
            reportEntity = reportRepository.findById(reportId)
                    .orElseThrow(()              ->              new
ResourceNotFoundException("Cannot create issue: attached report with
ID "
                            + reportId + " not found."));
        }

        String description = createRequest.description();

        IssueEntity issueEntity = IssueEntity.builder()
                .client(userEntity)
                .subject(createRequest.subject())
```

```
                .description(description)
                .report(reportEntity)
                .status(IssueStatus.OPEN)
                .build();

        issueEntity = issueRepository.save(issueEntity);

        MessageEntity messageEntity = MessageEntity.builder()
                .issue(issueEntity)
                .sender(userEntity)
                .report(reportEntity)
                .textContent(description)
                .build();

        messageRepository.save(messageEntity);

        return
issueMapper.issueEntityToIssueCreateResponseDTO(issueEntity);
    }

    public     Page<IssueGetResponseDTO>     getClientIssues(Pageable
pageable) {
        Integer clientId = getCurrentUserId();

        Return       issueRepository.findClientIssuesSorted(clientId,
pageable);
    }

    public    Page<MessageSendResponseDTO>    getIssueMessages(Integer
issueId, Pageable pageable) {
        Integer currentUserId = getCurrentUserId();

        IssueEntity issue = issueRepository.findById(issueId)
                .orElseThrow(()                 ->                 new
ResourceNotFoundException("Issue not found"));
```

```
        if (!issue.getClient().getId().equals(currentUserId)) {
            throw new AccessDeniedException("You do not have
permission to access this issue");
        }

        Page<MessageEntity> messageEntities =
messageRepository.findByIssueId(issueId, pageable);

        return messageEntities.map(
                messageEntity ->

messageMapper.messageEntityToMessageSendResponseDTO(messageEntity,
null)
        );
    }

    public void closeIssue(Integer issueId) {
        IssueEntity issueEntity = issueRepository.findById(issueId)
                .orElseThrow(() -> new
MyResponseStatusException(HttpStatus.NOT_FOUND, "Issue not found"));

        Integer expectedClientId = issueEntity.getClient().getId();

        if (!expectedClientId.equals(getCurrentUserId())) {
            throw new MyResponseStatusException(HttpStatus.NOT_FOUND,
"Issue not found");
        }

        issueEntity.setStatus(IssueStatus.CLOSED);
        issueRepository.save(issueEntity);

        ChatStatusDTO chatStatusDTO = new
ChatStatusDTO(expectedClientId, issueId, IssueStatus.CLOSED);
```

```
        broker.convertAndSend("/topic/chat-status/" + issueId,
chatStatusDTO);
    }

    public Page<ReportGetResponseDTO> getClientReports(PageRequest
pageRequest) {
        Integer clientId = getCurrentUserId();

        Page<ReportEntity> reportEntities =
reportRepository.findAllByUserId(clientId, pageRequest);

        return
reportEntities.map(reportMapper::reportEntityToReportGetResponseDTO)
;
    }

    public void deleteClientReportById(Integer reportId) {
        ReportEntity reportEntity =
reportRepository.findById(reportId)
                .orElseThrow(() -> new
ResourceNotFoundException("Report not found"));
        Integer userId = getCurrentUserId();

        if (!reportEntity.getUser().getId().equals(userId)) {
            throw new ResourceNotFoundException("Report not found");
        }
        reportRepository.deleteById(reportId);
    }

    public ReportCreateResponseDTO generatePDFAsync(String userId,
ReportCreateRequestDTO reportCreateRequestDTO) {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        byte[] pdf =
reportService.createPDFReport(reportCreateRequestDTO);
```

```
        Integer reportId = saveInDB(Integer.valueOf(userId), pdf);
        stopWatch.stop();
        log.info("Time for processing report: {} ms",
stopWatch.getTotalTimeMillis());

        return ReportCreateResponseDTO.builder()
                .reportStatus(ReportStatus.READY)
                .reportId(reportId)
                .build();
    }

    public ReportEntity getPDFReport(Integer reportId) {
        return reportRepository.findById(reportId)
                .orElseThrow(() -> new RuntimeException("Report not
found"));
    }
}
```

Лістинг А.9 – Клас SpecialistService на сервері — функціонал для спеціалістів сервісного центру

```
@Service
@RequiredArgsConstructor
@Slf4j
public class SpecialistService {
    private final ReportRepository reportRepository;
    private final IssueRepository issueRepository;
    private final MessageMapper messageMapper;
    private final MessageRepository messageRepository;
    private final UserRepository userRepository;
    private final SimpMessagingTemplate broker;
    public Page<IssueGetResponseDTO> getSpecialistIssues(Pageable
pageable) {
        Integer specialistId = getCurrentUserId();
```

```
        return
issueRepository.findSpecialistIssuesSorted(specialistId, pageable);
    }

    public IssueStatProjection getIssueStats() {
        return issueRepository.countStat(getCurrentUserId())
                .orElseThrow(() -> new
MyResponseStatusException(HttpStatus.NOT_FOUND, "User not found"));
    }

    public Page<MessageSendResponseDTO> getIssueMessages(Integer
issueId, Pageable pageable) {
        Integer currentUserId = getCurrentUserId();

        IssueEntity issue = issueRepository.findById(issueId)
                .orElseThrow(() -> new
ResourceNotFoundException("Issue not found"));

        if (issue.getAssignedSpecialist() == null) {
            UserEntity userEntity =
userRepository.findById(currentUserId)
                    .orElseThrow(() -> new
MyResponseStatusException(HttpStatus.NOT_FOUND, "User not found"));
            issue.setAssignedSpecialist(userEntity);
            issue.setStatus(IssueStatus.IN_PROGRESS);
            issueRepository.save(issue);
        } else if
(!issue.getAssignedSpecialist().getId().equals(currentUserId)) {
            throw new AccessDeniedException("You do not have
permission to access this issue");
        }

        Page<MessageEntity> messageEntities =
messageRepository.findByIssueId(issueId, pageable);
```

```
        return messageEntities.map(messageEntity ->

messageMapper.messageEntityToMessageSendResponseDTO(messageEntity,
null)
        );
    }

    @Transactional
    public void closeIssue(Integer issueId) {
        log.info("Closing issue {}", issueId);
        IssueEntity issueEntity = issueRepository.findById(issueId)
                .orElseThrow(() -> new
MyResponseStatusException(HttpStatus.NOT_FOUND, "Issue not found"));

        Integer expectedSpecialistId =
issueEntity.getAssignedSpecialist().getId();

        if (!expectedSpecialistId.equals(getCurrentUserId())) {
            log.warn("expected id: {} does not match actual id {}",
expectedSpecialistId, getCurrentUserId());
            throw new MyResponseStatusException(HttpStatus.NOT_FOUND,
"Issue not found");
        }

        issueEntity.setStatus(IssueStatus.CLOSED);
        issueRepository.save(issueEntity);
        ChatStatusDTO chatStatusDTO = new
ChatStatusDTO(expectedSpecialistId, issueId, IssueStatus.CLOSED);

        broker.convertAndSend("/topic/chat-status/" + issueId,
chatStatusDTO);
    }

    public ReportEntity getPDFReport(Integer issueId, Integer
reportId) {
```

```
        Integer specialistId = getCurrentUserId();

        boolean issueWithSpecExist =
issueRepository.existsByIdAndAssignedSpecialistId(issueId,
specialistId);
        if (!issueWithSpecExist) {
            throw new MyResponseStatusException(HttpStatus.NOT_FOUND,
                    "No issue with id " + issueId + "  and specialist
id: " + specialistId + "found");
        }

        boolean reportInThisIssue =
messageRepository.existsByIssueIdAndReportId(issueId, reportId);
        if (!reportInThisIssue)
            throw new MyResponseStatusException(HttpStatus.NOT_FOUND,
"Report not found in this issue");

        return reportRepository.findById(reportId)
                .orElseThrow(() -> new
MyResponseStatusException(HttpStatus.NOT_FOUND, "Report not found"));
    }
}
```

Лістинг А.10 – Клас UserService на сервері — оновлення профілів користувачів

```
@Service
@RequiredArgsConstructor
@Slf4j
public class UserService {
    private final UserRepository userRepository;
    private final Cache<String, ProfilePutRequestDTO>
profileUpdateCache;
    private final Cache<String, String> verificationCodeCache;
    private final EmailService emailService;
```

```
    private final PasswordEncoder passwordEncoder;
    private final Random random;
    private final UserMapper userMapper;

    public ProfilePutResponseDTO initiateProfileUpdate(ProfilePutRequestDTO dto) {
        if (dto.getEmail() != null && userRepository.existsByEmail(dto.getEmail())) {
            throw new MyResponseStatusException(HttpStatus.CONFLICT, "Already exists email", "Ця пошта вже використовується");
        }

        UserEntity userEntity = userRepository.findById(getCurrentUserId())
                .orElseThrow(() -> new MyResponseStatusException(HttpStatus.NOT_FOUND, "User not found"));

        String uuid = UUID.randomUUID().toString();

        if (dto.getPassword() != null) {
         dto.setPassword(passwordEncoder.encode(dto.getPassword()));
        }

        log.info("Name: {}, email: {}, password: {}", dto.getName(), dto.getEmail(), dto.getPassword());
        profileUpdateCache.put(uuid, dto);

        String targetEmail = dto.getEmail() == null ? userEntity.getEmail() : dto.getEmail();

        String code = String.format("%04d", random.nextInt(999999));
        verificationCodeCache.put(uuid, code);

        emailService.sendEmail(targetEmail, code, "Код для підтвердження зміни профілю");
```

```
        return new ProfilePutResponseDTO(uuid, "Код надіслано на
email");
    }
    public UserResponseDTO confirmProfileUpdate(ProfileConfirmDTO
profileConfirmDTO) {
        String uuid = profileConfirmDTO.uuid();
        Integer code = profileConfirmDTO.code();

        ProfilePutRequestDTO cachedDto =
profileUpdateCache.getIfPresent(uuid);
        if (cachedDto == null) {
            throw new
MyResponseStatusException(HttpStatus.BAD_REQUEST, "Update session
expired or invalid UUID");
        }
        log.info("cacheDTO: {}", cachedDto);
        log.info("{},{},{}", cachedDto.getName(),
cachedDto.getEmail(), cachedDto.getEmail());

        String cachedCode = verificationCodeCache.getIfPresent(uuid);
        if (cachedCode == null ||
!cachedCode.equals(String.format("%04d", code))) {
            throw new
MyResponseStatusException(HttpStatus.BAD_REQUEST, "Wrong code");
        }

        UserEntity userEntity =
userRepository.findById(getCurrentUserId())
                .orElseThrow(() -> new
MyResponseStatusException(HttpStatus.NOT_FOUND, "User not found"));

        if (cachedDto.getName() != null) {
            userEntity.setNickname(cachedDto.getName());
        }
```

```
        if (cachedDto.getEmail() != null) {
            userEntity.setEmail(cachedDto.getEmail());
        }
        if (cachedDto.getPassword() != null) {
            userEntity.setPassword(cachedDto.getPassword());
        }

        userEntity = userRepository.save(userEntity);

        profileUpdateCache.invalidate(uuid);
        verificationCodeCache.invalidate(uuid);

        log.info("Profile updated successfully for user:{}",
userEntity.getId());
        return userMapper.userEntityToUserResponseDTO(userEntity);
    }
}
```

Лістинг А.11 – GlobalExceptionHandler — Клас для обробки помилок на сервері

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(MyResponseStatusException.class)
    public                                ResponseEntity<ProblemDetail>
handleApiStatus(MyResponseStatusException ex,

HttpServletRequest request) {

        ProblemDetail pd = ProblemDetail.forStatus(ex.getStatus());
        pd.setTitle(ex.getTitle());
        pd.setDetail(ex.getDetail());
        pd.setInstance(URI.create(request.getRequestURI()));
```

```
        pd.setProperty("timestamp", Instant.now());

        log.warn("ProblemDetail returned: {}", pd);
        return ResponseEntity.status(ex.getStatus()).body(pd);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ProblemDetail> handleUnexpected(Exception
ex,

HttpServletRequest request) {
        ProblemDetail pd =
ProblemDetail.forStatus(HttpStatus.INTERNAL_SERVER_ERROR);
        pd.setTitle("Internal error");
        pd.setDetail(ex.getMessage());
        pd.setInstance(URI.create(request.getRequestURI()));
        pd.setProperty("timestamp", Instant.now());
        log.error("Unexpected error", ex);
        return ResponseEntity.internalServerError().body(pd);
    }
}
```

Лістинг А.12 – Фрагмент коду клієнтського сервісу CrossPlatformDataGatherService

```
package com.iafanasiev.syshelperdesktop.service.report;
import
com.iafanasiev.syshelperdesktop.dto.report.ReportCreateRequestDTO;
import com.iafanasiev.syshelperdesktop.dto.report.data.*;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import oshi.SystemInfo;
import oshi.hardware.*;
import oshi.software.os.*;
import java.util.List;
```

```
@Service
@Slf4j
public class CrossPlatformDataGatherService {

    private final SystemInfo systemInfo = new SystemInfo();

    public ReportCreateRequestDTO gatherSystemData() {
        HardwareAbstractionLayer hardware = systemInfo.getHardware();
        OperatingSystem os = systemInfo.getOperatingSystem();

        ReportCreateRequestDTO dto = ReportCreateRequestDTO.builder()
                .processor(getProcessorDTO(hardware))
                .memory(collectMemoryInfo(hardware))
                // ... інші методи збору даних ...
                .build();

        log.info("System report DTO successfully created: {}", dto);
        return dto;
    }

    public ProcessorDTO getProcessorDTO(HardwareAbstractionLayer
hardware) {
        CentralProcessor processor = hardware.getProcessor();
        CentralProcessor.ProcessorIdentifier id =
processor.getProcessorIdentifier();

        ProcessorDTO dto = new ProcessorDTO(
                id.getName(),
                id.getVendor(),
                id.getVendorFreq(),
                processor.getMaxFreq(),
                processor.getLogicalProcessorCount(),
                processor.getPhysicalProcessorCount(),
                processor.getPhysicalPackageCount(),
```

```
processor.getSystemCpuLoadBetweenTicks(processor.getSystemCpuLoadTic
ks())
        );

        log.debug("Processor DTO created: {}", dto);
        return dto;
    }

    private    MemoryDTO    collectMemoryInfo(HardwareAbstractionLayer
hardware) {
        GlobalMemory memory = hardware.getMemory();

        List<PhysicalMemoryDTO>          physicalMemories          =
memory.getPhysicalMemory().stream()
                .map(pm -> new PhysicalMemoryDTO(
                        pm.getBankLabel(),
                        pm.getCapacity(),
                        pm.getClockSpeed(),
                        pm.getManufacturer(),
                        pm.getMemoryType(),
                        pm.getPartNumber()
                ))
                .toList();

        VirtualMemory vm = memory.getVirtualMemory();
        VirtualMemoryDTO              virtualMemoryDTO              =
VirtualMemoryDTO.builder()
                .swapTotal(vm.getSwapTotal())
                .swapUsed(vm.getSwapUsed())
                .swapPagesIn(vm.getSwapPagesIn())
                .swapPagesOut(vm.getSwapPagesOut())
                .virtualInUse(vm.getVirtualInUse())
                .virtualMax(vm.getVirtualMax())
                .build();
```

```
        return new MemoryDTO(
                memory.getTotal(),
                memory.getAvailable(),
                memory.getPageSize(),
                physicalMemories,
                virtualMemoryDTO
        );
    }
}
```

Лістинг А.13 – Фрагмент JavaFX контролера ClientCenterViewController

```
public class ClientCenterViewController {
    private final SystemReportService systemReportService;
    private final CrossPlatformDataGatherService
crossPlatformDataGather;
    private final UserSession userSession;
    private final ClientService clientService;

    @FXML
    public Label reportStatusLbl;
    @FXML
    public Button downloadReportBtn;
    @FXML
    public Button generateReportBtn;

    private final ExecutorService executorService =
Executors.newSingleThreadExecutor();
    private Integer readyReportId;

    @FXML
    public void handleGenerateReportBtn() {
        executorService.execute(this::processReportGeneration);
    }
```

```
    private void processReportGeneration() {
        this.readyReportId = null;
        Platform.runLater(()   ->   reportStatusLbl.setText("Генерація
звіту..."));

        ReportCreateRequestDTO        reportCreateRequestDTO        =
crossPlatformDataGather.gatherSystemData();
        ResponseStatusDTO<ReportCreateResponseDTO> response =

systemReportService.generateSystemReport(userSession.getToken(),
reportCreateRequestDTO);

        ReportCreateResponseDTO responseDTO = response.getPayload();
        handleReportStatus(responseDTO);
    }

    private void handleReportStatus(ReportCreateResponseDTO status) {
        Platform.runLater(() -> {
            if (status.reportStatus() == ReportStatus.READY) {
                reportStatusLbl.setText("Звіт готовий");
                downloadReportBtn.setDisable(false);
                this.readyReportId = status.reportId();
            } else if (status.reportStatus() == ReportStatus.ERROR) {
                reportStatusLbl.setText("Помилка генерації звіту");
            }
        });
    }

    @FXML
    public void handleDownloadReportBtn() {
        ResponseStatusDTO<FileDownloadResponseDTO>
reportGetResponseDTO =

systemReportService.getPDFReport(userSession.getToken(),
readyReportId);
```

```
        if (reportGetResponseDTO.isSuccess()) {
            // логіка збереження pdf...
        } else {
            reportStatusLbl.setText("Помилка завантаження звіту");
        }
    }

    @PreDestroy
    public void shutdownScheduler() {
        executorService.shutdownNow();
    }
}
```

Лістинг А.14 – Фрагмент коду JavaFX контролера SpecialistCenterViewController

```
@Component
@Slf4j
public class SpecialistCenterViewController {
    private final SpecialistService specialistService;
    @FXML
    private Label assignedCountLbl;
    @FXML
    private Label openCountLbl;

    public SpecialistCenterViewController(SpecialistService
specialistService) {
        this.specialistService = specialistService;
    }

    @FXML
    public void initialize() {
        log.info("Initialize RightPanelViewController");
```

```
        IssueStatsResponseDTO issueStatsResponseDTO =
specialistService.getSpecialistStats();
        updateCounters(issueStatsResponseDTO);
    }

    public void updateCounters(IssueStatsResponseDTO
issueStatsResponseDTO) {
        Platform.runLater(() -> {

assignedCountLbl.setText(String.valueOf(issueStatsResponseDTO.assign
ed()));

openCountLbl.setText(String.valueOf(issueStatsResponseDTO.open()));
        });
    }
}
```

Лістинг А.15 – Фрагмент коду JavaFX контролера ChatViewController

```
@Component
@RequiredArgsConstructor
@Slf4j
public class ChatViewController implements RequiresWSConnection {
    private final MainController mainController;
    private final SceneManager sceneManager;
    private final ClientService clientService;
    private final UserSession userSession;
    private final SystemReportService systemReportService;
    private final SpecialistService specialistService;
    private final ChatService chatService;
    private final ObservableList<MessageSendResponseDTO> messages =
FXCollections.observableArrayList();
    @FXML
    public Button closeChatBtn;
    @FXML
    public Label statusLbl;
```

```
    @FXML
    public Button sendButton;
    @FXML
    public TextArea messageInput;
    @FXML
    private ListView<MessageSendResponseDTO> messageListView;

    private Integer issueId;

    @FXML
    public void initialize() {
        log.info("Initializing Chat View Controller");
        messageListView.setItems(messages);

        messageListView.setCellFactory(listView -> new ListCell<>() {
            @Override
            protected  void  updateItem(MessageSendResponseDTO  item,
boolean empty) {
                super.updateItem(item, empty);
                if (empty || item == null) {
                    setGraphic(null);
                } else {
                    MessageItemController controller =

sceneManager.loadComponent(SceneItemType.MESSAGE_ITEM,
this::setGraphic, false);

                    controller.setData(item, userSession.getUserId(),
reportId -> handleDownloadReport(reportId));
                }
            }
        });
    }

    public void initChat(IssueGetResponseDTO issue) {
```

```
        log.info("initChat invoked");
        log.info("Issue: {}. IssueId: {}", issue, issue.Id());
        this.issueId = issue.Id();

        if (issue.status().equals(IssueStatus.CLOSED)) {
            applyClosedChatState();
        } else {
            chatService.subscribe(issueId,
this::handleReceiveMessage, this::handleChatStatusMessage);
        }
        List<MessageSendResponseDTO>          messageList          =
getChatMessageList();
        messages.clear();
        messages.addAll(messageList);

        Platform.runLater(() -> {
            if (!messages.isEmpty()) {
                messageListView.scrollTo(messages.size() - 1);
            }
        });
    }

    @FXML
    public void handleSendBtn() {
        String textContent = messageInput.getText().trim();
        if (textContent.isBlank()) return;

        String     tempId     =     chatService.sendMessage(issueId,
textContent);

        MessageSendResponseDTO                pending                =
MessageSendResponseDTO.builder()
                .tempId(tempId)
                .senderId(userSession.getUserId())
                .textContent(textContent)
```

```
                .sentAt(String.valueOf(LocalDateTime.now()))
                .build();

        messages.add(pending);
        messageInput.clear();
        Platform.runLater(() ->
messageListView.scrollTo(messages.size() - 1));
    }

    @FXML
    public void handleBack() {
        chatService.disconnect();
        if (Objects.requireNonNull(userSession.getRole()) ==
Role.CLIENT) {

sceneManager.loadComponent(SceneViewType.CLIENT_CENTER_VIEW,
mainController.mainView::setCenter);
        } else if (userSession.getRole() == Role.SPECIALIST) {

sceneManager.loadComponent(SceneViewType.SPECIALIST_CENTER_VIEW,
mainController.mainView::setCenter);
        }
    }

    @FXML
    public void handleCloseChat() {
        closeChatBtn.setDisable(true);
        ResponseStatusDTO<Void> responseStatusDTO = null;

        if (userSession.getRole() == Role.CLIENT) {
            responseStatusDTO = clientService.closeChat(issueId);
        } else if (userSession.getRole() == Role.SPECIALIST) {
            responseStatusDTO = specialistService.closeChat(issueId);
        } else {
```

```
            throw new IllegalStateException("Unexpected role: " +
userSession.getRole());
        }

        if (responseStatusDTO.isSuccess()) {
            chatService.unsubscribe();
            applyClosedChatState();

            updateLblTextAndColorTemporary(statusLbl, "Чат закрито
успішно", Color.GREEN, 4);
        } else {
            closeChatBtn.setDisable(false);
            updateLblTextAndColorTemporary(statusLbl,
responseStatusDTO.getMessage(), Color.RED, 6);
        }
    }
}
```

Лістинг А.16 – Конфігурація Spring Security WebSocket

```
@Configuration
@EnableWebSocketSecurity
@RequiredArgsConstructor
@Slf4j
public class WebSocketSecurityConfig {
    private final IssueRepository issueRepository;

    @Bean
    AuthorizationManager<Message<?>> messageAuthorizationManager(
            MessageMatcherDelegatingAuthorizationManager.Builder
messages) {
        messages.simpTypeMatchers(
                SimpMessageType.CONNECT,
                SimpMessageType.DISCONNECT,
                SimpMessageType.UNSUBSCRIBE
        ).authenticated();
```

```
messages.simpSubscribeDestMatchers("/user/**").hasRole(Role.CLIENT.n
ame());

        messages.simpDestMatchers("/topic/chat/**",     "/topic/chat-
status/**").access(this::hasAccessToChat);
        messages.simpSubscribeDestMatchers("/topic/chat/**",
"/topic/chat-status/**").access(this::hasAccessToChat);

messages.simpMessageDestMatchers("/app/chat/**").authenticated();
        messages.anyMessage().denyAll();

        return messages.build();
    }

    private                                    AuthorizationDecision
hasAccessToChat(Supplier<Authentication> authentication,

MessageAuthorizationContext<?> context) {
        log.info("Authentication:  {}.  MessageAuthorizationContext:
{}", authentication.get(), context);
        Message<?> msg = context.getMessage();

        String              dest              =              (String)
msg.getHeaders().get("simpDestination");

        Integer                        issueId                       =
Integer.valueOf(dest.substring(dest.lastIndexOf('/') + 1));
        Integer                        userId                        =
Integer.valueOf(authentication.get().getName());

        boolean allowed = checkAccess(userId, issueId);
        log.info("Allowed access: {}", allowed);

        return new AuthorizationDecision(allowed);
    }
```

```
    private boolean checkAccess(Integer userId, Integer issueId) {
        return   issueRepository.issueExistWithSpecifiedUser(issueId,
userId);
    }
}
```

Лістинг А.17 – Код класу JwtAuthenticationFilter

```
@Component
@Slf4j
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    public static final String AUTHORIZATION_HEADER = "Authorization";
    public static final String BEARER_PREFIX = "Bearer";
    private final JwtService jwtService;

    @Override
    protected   void   doFilterInternal(@NonNull   HttpServletRequest
request,
                                        @NonNull      HttpServletResponse
response,
                                        @NonNull FilterChain filterChain)
throws ServletException, IOException {
        final             String             authHeader             =
request.getHeader(AUTHORIZATION_HEADER);

        if          (authHeader          ==          null          ||
!authHeader.startsWith(BEARER_PREFIX)) {
            log.info("Without header or bearer");
            filterChain.doFilter(request, response);
            return;
        }

        final                String                jwt                =
authHeader.substring(BEARER_PREFIX.length() + 1);
        final String userId = jwtService.extractUsername(jwt);
```

```
        log.debug("Process JWT for user: {}", userId);

        Authentication                authentication                =
SecurityContextHolder.getContext().getAuthentication();

        if (userId != null && authentication == null) {
            log.debug("Start validation of token");
            if (jwtService.isTokenValid(jwt)) {
                log.info("Token valid for user: {}", userId);
                String role = jwtService.extractAuthority(jwt);

                UserDetails userDetails = UserEntity.builder()
                        .id(Integer.valueOf(userId))
                        .role(Role.valueOf(role))
                        .build();

                UsernamePasswordAuthenticationToken
authenticationToken = new UsernamePasswordAuthenticationToken(
                        userDetails,
                        null,
                        userDetails.getAuthorities()
                );
                authenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authenticationT
oken);
            } else {
                log.warn("Invalid token for user: {}", userId);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

Лістинг А.18 – Конфігурація Spring Security

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {
    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final AuthenticationProvider authenticationProvider;

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
                .csrf(AbstractHttpConfigurer::disable)
                .headers(header                                          ->
header.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin
))
                .cors(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(request -> request
                        .requestMatchers("/api/auth/**").permitAll()
.requestMatchers("/api/users/**").authenticated()

.requestMatchers("/api/clients/**").hasRole(Role.CLIENT.name())

.requestMatchers("/api/specialists/**").hasRole(Role.SPECIALIST.name
())
.requestMatchers("/api/admin/**").hasRole(Role.ADMIN.name())
                        .requestMatchers("/ws").authenticated()
                        .anyRequest().permitAll())
                .sessionManagement(manager                               ->
manager.sessionCreationPolicy(STATELESS))
                .authenticationProvider(authenticationProvider)
                .addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class);

        return http.build();
```

```
    }
}
```

Лістинг А.19 – Сценарій навантажувального тестування Gatling

```
package com.iafanasiev;

import io.gatling.javaapi.core.*;
import io.gatling.javaapi.http.*;
import static io.gatling.javaapi.core.CoreDsl.*;
import static io.gatling.javaapi.http.HttpDsl.*;

public class CreateIssueSimulation extends Simulation {

    HttpProtocolBuilder httpProtocol = http
            .baseUrl("http://ec2-user@ec2-3-84-249-103.compute-
1.amazonaws.com:8081")
            .acceptHeader("application/json");

    FeederBuilder.Batchable<String>             feeder             =
csv("data/users.csv").circular();

    static final String ISSUE_BODY = Utils.loadReportJson();

    ScenarioBuilder  scn  =  scenario("Login,  Then  Create  Multiple
Reports If Authenticated")
            .feed(feeder)
            .exec(
                    http("User Login")
                            .post("/api/auth/sign-in")
                            .header("Content-Type",
"application/json")
                            .body(StringBody(session                ->
String.format("{\"email\":\"%s\", \"password\":\"%s\"}",
                                    session.getString("email"),
session.getString("password"))))
```

```
                        .check(status().is(200))

.check(jsonPath("$.token").saveAs("jwtToken"))
            )
            .exitHereIfFailed()
            .repeat(20, "n").on(
                    exec(
                            http("Create Report #{n}")
                                    .post("/api/clients/me/reports")
                                    .header("Authorization",  session
-> "Bearer " + session.getString("jwtToken"))
                                    .header("Content-Type",
"application/json")
                                    .body(StringBody(ISSUE_BODY))
                                    .check(status().is(201))
                    )
                            .pause(1)
            );

    {
        setUp(
                scn.injectOpen(atOnceUsers(50))
        ).protocols(httpProtocol);
    }
}
```

ДОДАТОК Б
ПРИКЛАДИ ІНТЕРФЕЙСУ КОРИСТУВАЧА

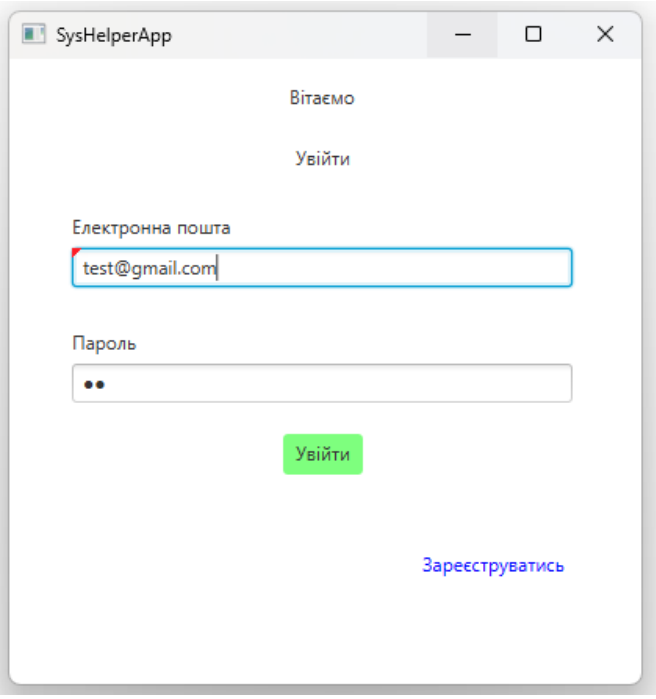


Рисунок Б.1 – Вікно входу до системи

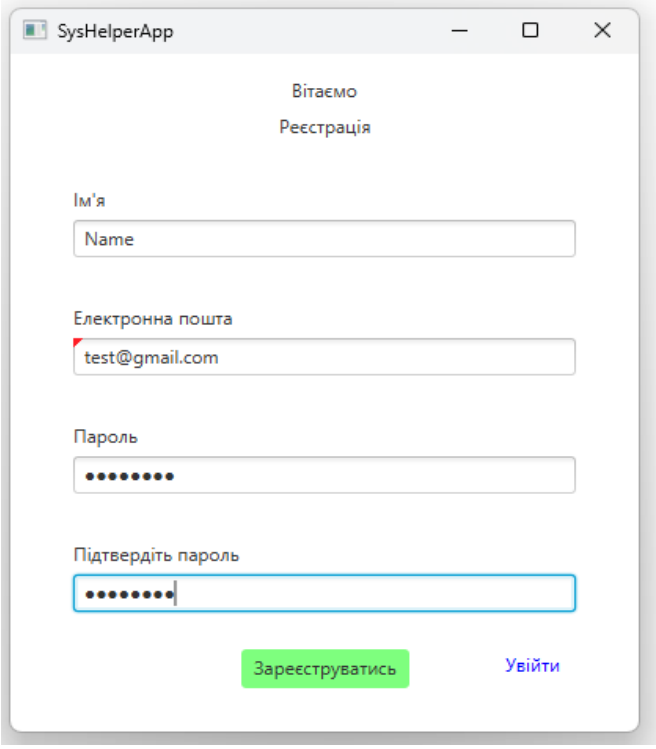


Рисунок Б.2 – Вікно реєстрації нового користувача

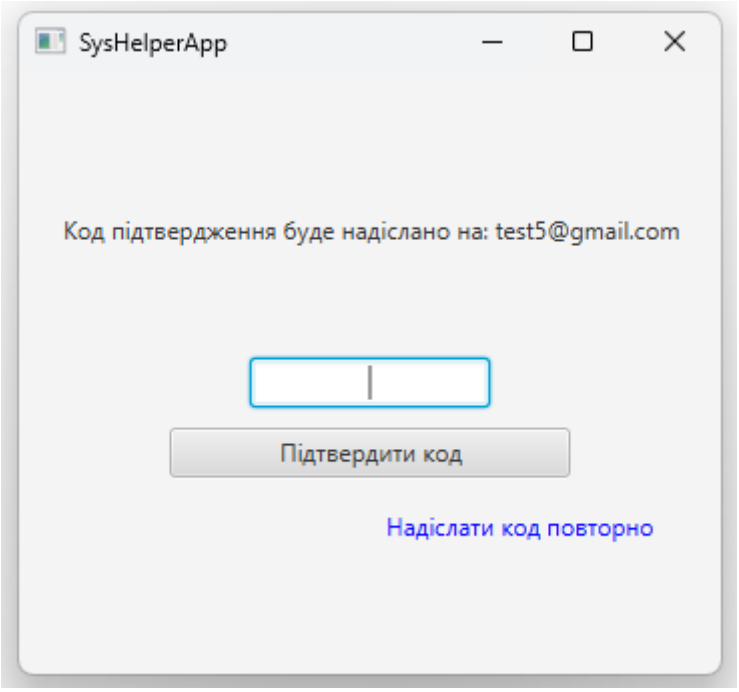


Рисунок Б.3 – Вікно підтвердження електронної пошти при реєстрації користувача

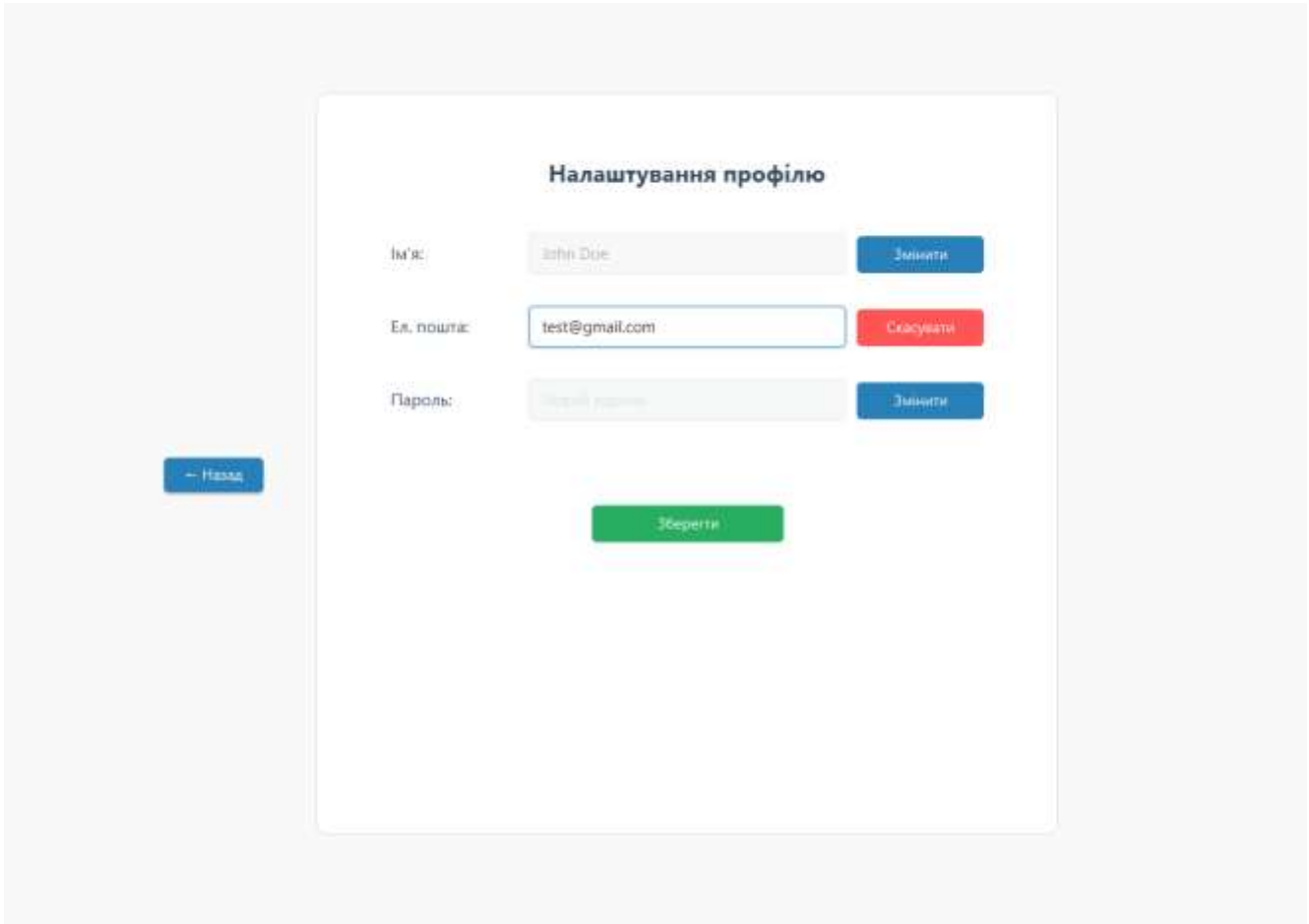


Рисунок Б.4 – Вікно редагування профілю користувача

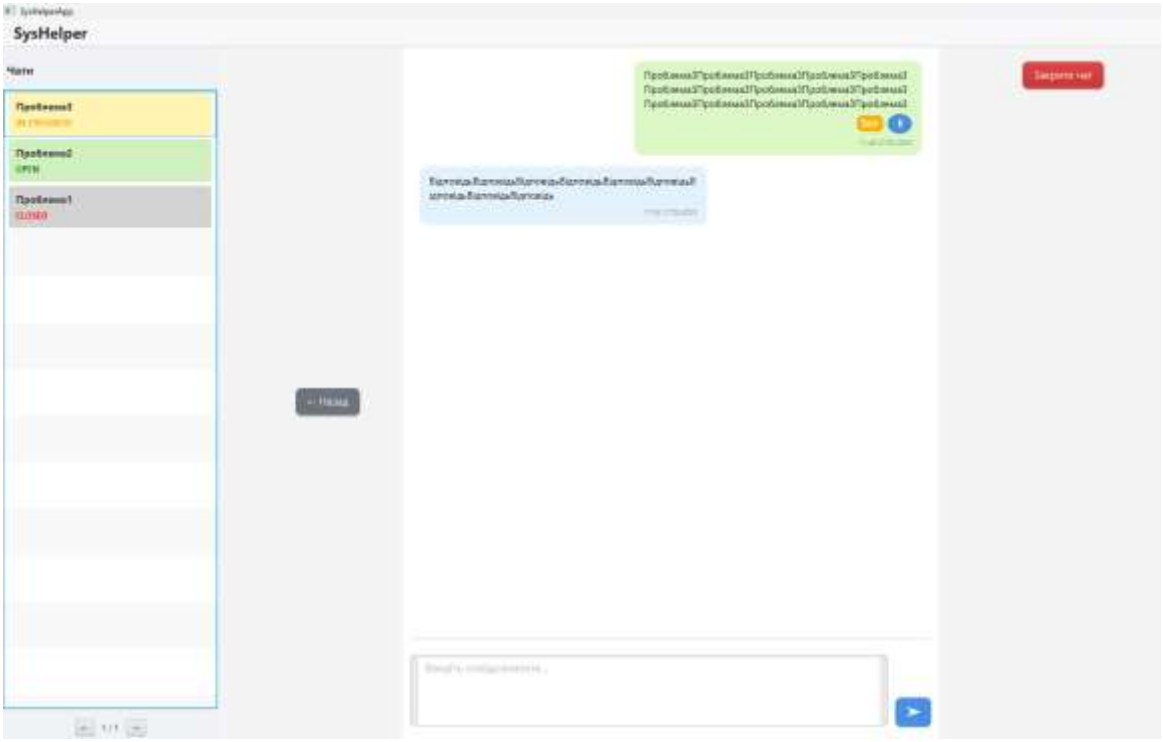


Рисунок Б.5 – Вікно чату: спільний інтерфейс для клієнта та спеціаліста

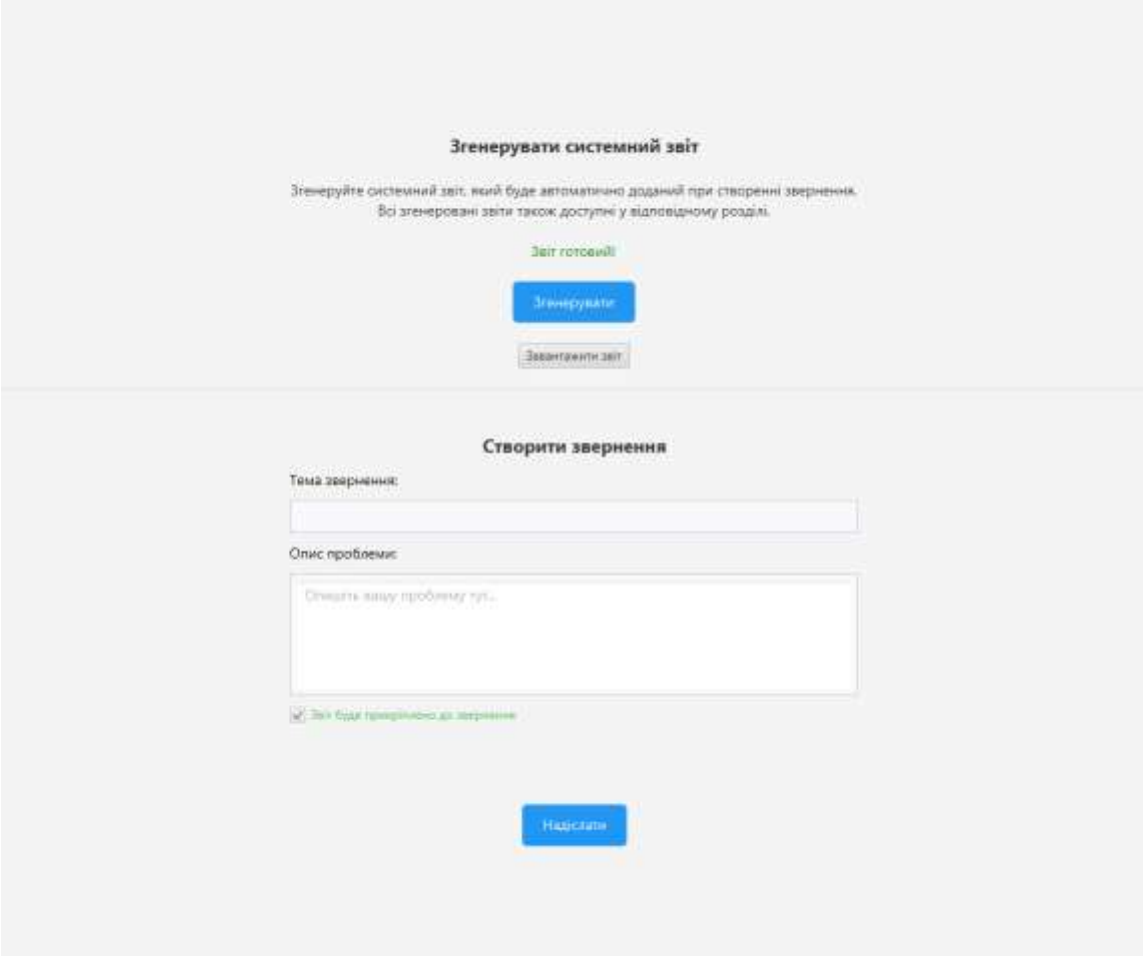


Рисунок Б.6 – Вікно "Кабінет користувача"

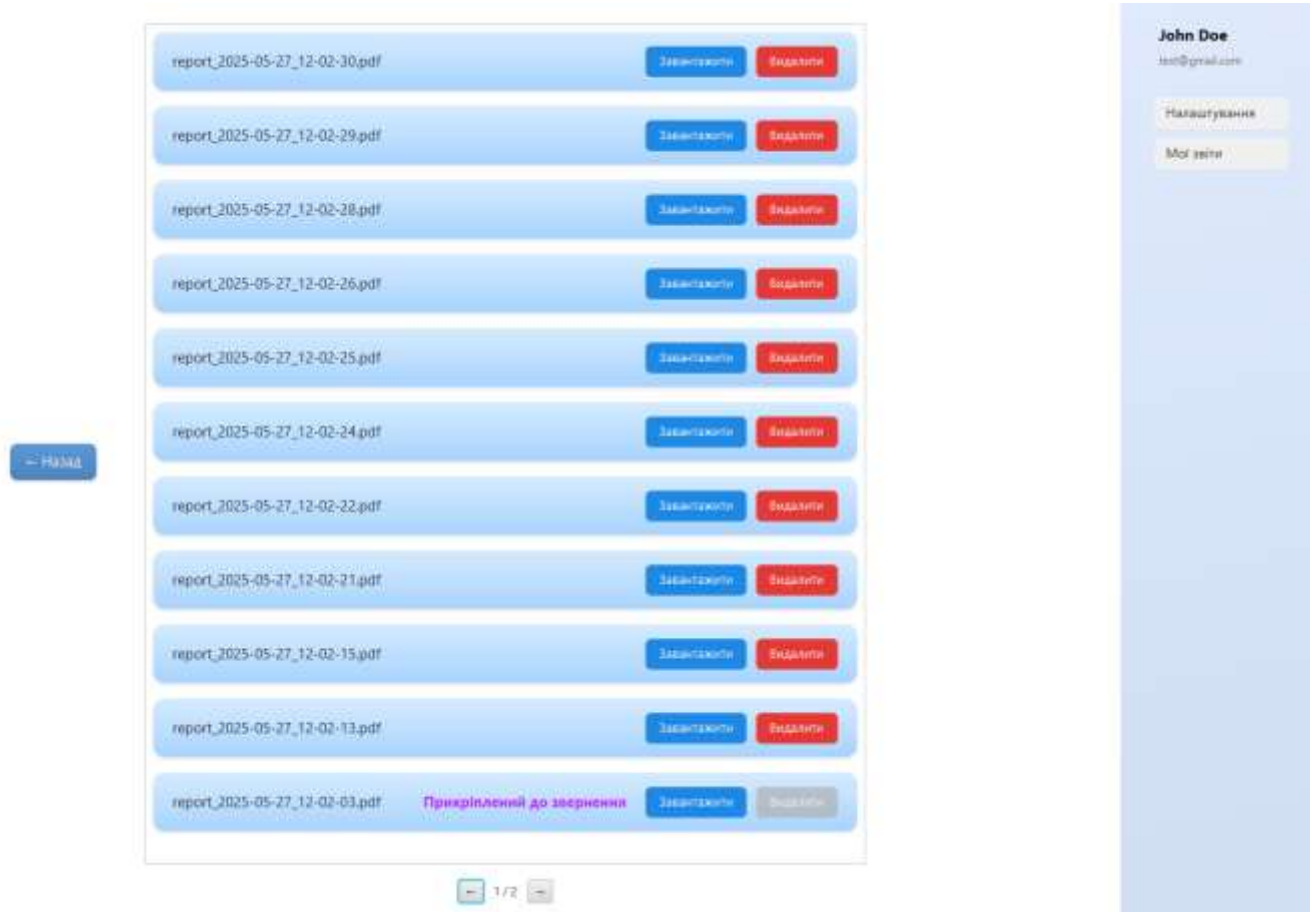


Рисунок Б.7 – Вікно перегляду звітів клієнта

Рисунок Б.8 – Головний екран спеціаліста

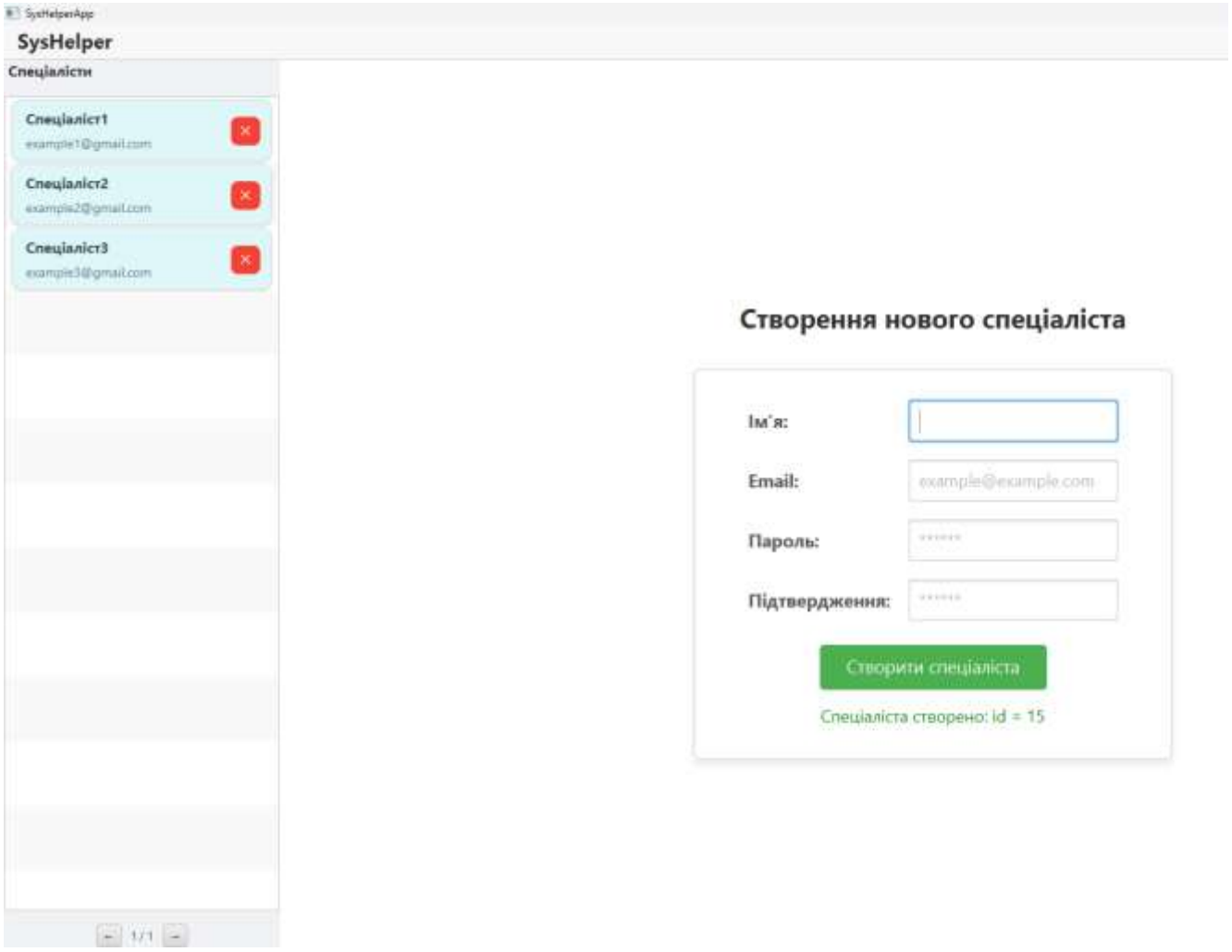


Рисунок Б.9 – Інтерфейс адміністратора

ДОДАТОК В
ПРИКЛАД ЗВІТУ ПРО СИСТЕМНІ ПАРАМЕТРИ

System Report

Creation Date and Time: 2025-05-20 16:49:14

Operating System Information

Operating System:

- Name = Windows
- Version = 11 (, build: 26100)
- Manufacturer = Microsoft
- Architecture = 64-bit
- Boot Time = 2025-05-15 08:29:53
- Uptime = 5 days, 08:19:40

Computer System

- Manufacturer = LENOVO
- Model = 82FE
- BIOS:
  - Manufacturer = LENOVO
  - Name = FKCN41WW(V3.04)
  - Version = LENOVO - 1
  - Release Date = 2022-03-04
- Baseboard:
  - Manufacturer = LENOVO
  - Model = LNVNB161216
  - Version = No DPK

Processor

- Name = 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
- Vendor = GenuineIntel
- Base Frequency = 3.00 GHz
- Max Frequency = 3.00 GHz
- Logical Processors = 4
- Physical Processors = 2
- Physical Packages = 1
- CPU Load = 13.10%

Рисунок В.1 – Зведена інформація про операційну систему, комп’ютерну систему та процесор

Memory

Total = 7.79 GB
Available = 774.12 MB
Page Size = 4.00 KB

Physical Memory

Bank Label = BANK 0
Capacity = 4.00 GB
Clock Speed = 3200000000 MHz
Manufacturer = SK Hynix
Memory Type = DDR4
Part Number = HMA851S6CJR6N-XN

Bank Label = BANK 0
Capacity = 4.00 GB
Clock Speed = 3200000000 MHz
Manufacturer = SK Hynix
Memory Type = DDR4
Part Number = HMA851S6CJR6N-XN

Virtual Memory

Swap Total = 14.49 GB
Swap Used = 2.95 GB
Swap Pages In = 393574743
Swap Pages Out = 177002560
Virtual In Use = 17.66 GB
Virtual Max = 22.28 GB

Graphics Cards

Name = Intel(R) UHD Graphics
Vendor = Intel Corporation
Device ID = VideoController1
Driver Version = 30.0.101.1960
VRAM = 1.00 GB

Рисунок В.2 – Параметри оперативної пам’яті, віртуальної пам’яті та графічної підсистеми

Disk Stores

Model = SAMSUNG MZVLB512HBJQ-000L2 (Standard disk drives)
Name = \\.\PHYSICALDRIVE0
Size = 476.94 GB
Reads = 38251652
Bytes Read = 1.43 TB
Writes = 9795698
Bytes Written = 1.33 TB
Transfer Time = 13206116 ms
Current Queue Length = 0
Timestamp = 1747752552879

Partitions

Name = GPT: Basic Data
Identification = Disk #0, Partition #1
Type = GPT: Basic Data
Mount Point = C:\
Size = 116.15 GB
Major = 0
Minor = 1

Name = GPT: Basic Data
Identification = Disk #0, Partition #3
Type = GPT: Basic Data
Mount Point = D:\
Size = 359.75 GB
Major = 0
Minor = 3

Рисунок В.3 – Інформація про дискові накопичувачі та розділи

File Stores

Name = Local Fixed Disk (C:)
Mount = C:\
Type = NTFS
Total Space = 116.15 GB
Free Inodes = 0
Total Inodes = 0
Free Space = 7.00 GB

Name = Local Fixed Disk (D:)
Mount = D:\
Type = NTFS
Total Space = 359.75 GB
Free Inodes = 0
Total Inodes = 0
Free Space = 111.50 GB

Power Sources

Name = System Battery
Device Name = L19C3PF3
Manufacturer = Celxpert
Chemistry = Li-I
Cycle Count = 443
Current Capacity = 44540 mWh
Max Capacity = 44800 mWh
Design Capacity = 56500 mWh
Voltage = 12.78 V
Power Usage Rate = 0.00 mW
Temperature = 0.00 °C
Time Remaining Estimated = -0.00 hours
Time Remaining Instant = -0.00 hours
Remaining Capacity = 100 %
Is Charging = No
Is Discharging = No
Is Power On Line = Yes
Manufacture Date = N/A

Sensors

CPU Temperature = 0.00 °C
CPU Voltage = 0.00 V

Рисунок В.4 – Стан файлових розділів, акумулятора та датчиків системи

Sound Cards

Name =
Driver Version = IntcDMic.sys 10.29.0.6367

Name = Microsoft Microsoft Bluetooth A2dp Source
Driver Version = BthA2dp.sys 10.0.26100.1

Name = Realtek Semiconductor Corp. Realtek High Definition Audio(SST)
Driver Version = RTKVHD64.sys 6.0.9250.1

Name = Microsoft Microsoft Bluetooth A2dp Sink
Driver Version = BthA2dp.sys 10.0.26100.1

Name = Microsoft Microsoft Bluetooth A2dp Source
Driver Version = BthA2dp.sys 10.0.26100.1

USB Devices

Name = Intel(R) USB 3.10 eXtensible Host Controller - 1.20 (Microsoft)
Product ID = 0xa0ed
Vendor = Generic USB xHCI Host Controller
Vendor ID = 0x8086

Name = Intel(R) USB 3.10 eXtensible Host Controller - 1.20 (Microsoft)
Product ID = 0x9a13
Vendor = Generic USB xHCI Host Controller
Vendor ID = 0x8086

Рисунок В.5 – Список аудіо-пристроїв і підключених USB-пристроїв

Network Interfaces

Name = wireless_3
Display Name = Intel(R) Wireless-AC 9560-QoS Packet Scheduler-0000
Alias = Wi-Fi-QoS Packet Scheduler-0000
Operational Status = UP
Type = 71
MTU = 1500
Connector Present = No
Bytes Received = 559.24 MB
Bytes Sent = 201.74 MB
Packets Received = 569904
Packets Sent = 480900
Speed = 866.70 Mbps

Name = wireless_32768
Display Name = Intel(R) Wireless-AC 9560
Alias = Wi-Fi
Operational Status = UP
Type = 71
MTU = 1500
Connector Present = Yes
Bytes Received = 559.24 MB
Bytes Sent = 201.74 MB
Packets Received = 569904
Packets Sent = 480900
Speed = 866.70 Mbps

Network Parameters

IPv4 Default Gateway = 192.168.0.1
IPv6 Default Gateway = fe80::362c:c4ff:fe35:d135

DNS Servers

- 192.168.0.1

Internet Protocol Statistics

TCP Connections Established = 62
TCP Segments Sent = 29325120
TCP Segments Received = 29377595
UDP Datagrams Sent = 1567462
UDP Datagrams Received = 3190779
Total Connections = 170
Established Connections = 61
Listening Connections = 53

Рисунок В.6 – Характеристики мережевих інтерфейсів і мережеві статистики

Displays (EDID Hex)

Display 1:

00FFFFFFFFFFF0030AE229000000000291D0104A51F1178032865975954 8E271E50
5400000001010101010101010101010101010101363680A0703820403020A60035AD10
000018000000FE000A2020202020202020202020000000FE004C454E4F564F0A202
020202020000000FE004C454E31343046484420202020000800000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000

Рисунок В.7 – Шістнадцятковий дамп EDID для дисплеїв

Filtered Processes

Process ID = 17788
Up Time = 0 days, 07:50:59
Resident Set Size = 2.61 GB
CPU Load Cumulative = 25.66%
Thread Count = 136
State = RUNNING
Path = C:\Program Files\JetBrains\IntelliJ IDEA 2024.3.5\bin\idea64.exe

Process ID = 16516
Up Time = 0 days, 07:50:48
Resident Set Size = 476.63 MB
CPU Load Cumulative = 1.24%
Thread Count = 50
State = RUNNING
Path = C:\Program Files\JetBrains\IntelliJ IDEA 2024.3.5\jbr\bin\java.exe

Process ID = 18392
Up Time = 0 days, 00:00:14
Resident Set Size = 313.91 MB
CPU Load Cumulative = 97.73%
Thread Count = 56
State = RUNNING
Path = C:\Program Files\Java\jdk-21\bin\java.exe

Process ID = 15528
Up Time = 0 days, 00:00:14
Resident Set Size = 93.63 MB
CPU Load Cumulative = 14.85%
Thread Count = 28
State = RUNNING
Path = C:\Program Files\Java\jdk-21\bin\java.exe

Process ID = 4812
Up Time = 0 days, 01:22:09
Resident Set Size = 7.54 MB
CPU Load Cumulative = 18.97%
Thread Count = 16
State = RUNNING
Path = C:\Program Files\Google\Chrome\Application\chrome.exe

Рисунок В.8 – Перелік ресурсомістких процесів, відфільтрованих за навантаженням

РОЗРОБКА ПРОГРАМНОГО КОМПЛЕКСУ МОНІТОРИНГУ СИСТЕМНИХ ПАРАМЕТРІВ ТА ОБЛІКУ КЛІЄНТСЬКИХ ЗВЕРНЕНЬ ДЛЯ СЕРВІСНОГО ЦЕНТРУ

Виконав:
студент КНТ-521
Афанасьєв І.І.

Актуальність теми

- Сервісні центри отримують великий потік звернень щодо несправностей ПК та ПЗ.
- Дистанційна діагностика потребує точних системних даних і швидкого каналу спілкування.
- Відсутність єдиного інструмента «збір параметрів → PDF-звіт → онлайн-чат» збільшує час обробки звернення.

Мета та завдання

- Мета – створити крос-платформений комплекс, що автоматизує збір системних параметрів ПК, формує PDF-звіт і забезпечує інтерактивний чат між клієнтом та сервісним центром.
- Аналітичне завдання – дослідити наявні рішення й обрати оптимальний технологічний стек.
- Проєктні завдання – спроектувати архітектуру, базу даних і ключові модулі клієнта та сервера.
- Реалізаційні завдання – розробити десктоп-клієнт (JavaFX) і сервер (Spring Boot), інтегрувати OSHI, PDFBox та WebSocket-чат.
- Перевірочні завдання – підготувати сценарії тестування, виконати функціональні, навантажувальні тести та оформити інструкцію користувача.

Обраний технологічний стек

- Java 21 – основна мова розробки клієнта й сервера
- JavaFX + FXML – UI десктопного застосунку
- Spring Boot 3 – REST-сервер і бізнес-логіка
- Spring Security + JWT – автентифікація та авторизація
- WebSocket / STOMP – чат у реальному часі
- PostgreSQL – реляційна база даних
- OSHI-core – збір системних параметрів ПК
- Apache PDFBox – створення PDF-звітів
- Gatling – навантажувальне тестування

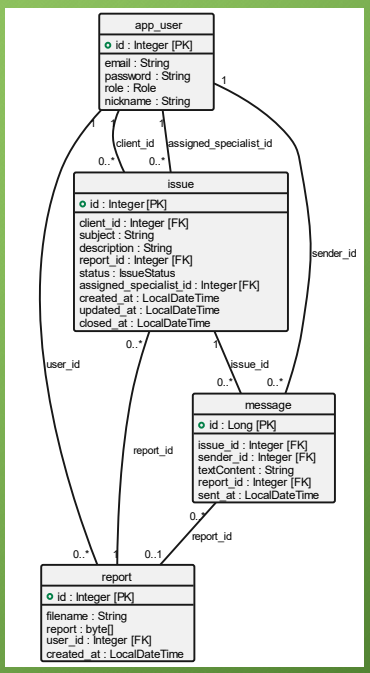
Архітектура системи

- Двошарова модель «клієнт - сервер»: десктоп-клієнт JavaFX і сервер Spring Boot.
- Клієнтський застосунок спілкується з сервером через REST API (JSON) для CRUD-операцій та через WebSocket/STOMP для обміну повідомленнями в чаті.
- На сервері: REST-контролери, сервісний шар (AuthService, ClientService, SpecialistService, AdminService), WebSocket-брокер і глобальний фільтр JWT.
- PostgreSQL зберігає користувачів, звернення, повідомлення та PDF-звіти (BLOB-поле).
- Усі запити підписуються JWT-токеном; WebSocket-сесія авторизується тим самим токеном, що забезпечує статичність сеансу без серверних сесій.

База даних

Головні таблиці:

- APP_USER
- ISSUE
- MESSAGE
- REPORT



Ключові програмні модулі

- AuthService + JWT – реєстрація, автентифікація, перевірка ролей і випуск токенів
- CrossPlatformDataGatherService (клієнт) – збір системних параметрів через OSHI-core
- ReportService / Apache PDFBox – формування PDF-звіту й збереження у REPORT (BLOB)
- ChatService + Spring WebSocket/STOMP – двобічний обмін повідомленнями в реальному часі

Business-сервіси:

- – ClientService – створення звернень, історія чатів, робота зі звітами
- – SpecialistService – приймання й обробка звернень, статистика, закриття чатів
- – AdminService – керування обліковими записами спеціалістів

Інтерфейс користувача

- Єдина форма входу (реєстрація для клієнта з підтвердженням e-mail, логін для всіх ролей)
- Клієнт – «Кабінет користувача»: генерація звіту, створення звернення, список чатів і розділ «Мої звіти»
- Спеціаліст – «Центр звернень»: статистика призначених і доступних чатів, історія повідомлень
- Адміністратор – «Управління персоналом»: список спеціалістів, форма додавання

У всіх вікно ділиться на такі частини:

- Ліва панель — навігація по чатах або спеціалістах;
- Центральна частина — основний робочий екран;
- Права панель — профіль і налаштування

Генерація системного звіту

1. Користувач натискає «Згенерувати звіт»;
2. клієнтський сервіс OSHI збирає параметри CPU, RAM, дисків, мережі, датчиків;
3. зібрані дані формуються у JSON-структуру та передаються на сервер через REST-запит;
4. ReportService перетворює отриману інформацію на PDF-документ за допомогою Apache PDFBox і зберігає його у таблиці REPORT як BLOB разом із метаданими;
5. у відповідь клієнт отримує ідентифікатор звіту; за цим ID можна одразу завантажити PDF або автоматично прикріпити його до нового звернення.

Захист і безпека даних

- Аутентифікація та авторизація основані на JWT-токенах. Кожен запит REST і кожна WebSocket-сесія супроводжуються дійсним токеном (окрім логіну і реєстрації).
- Фільтри Spring Security перехоплюють запити, перевіряють підпис токена та розподіляють доступ згідно з роллю користувача.
- Паролі зберігаються лише у вигляді хешів BCrypt, що унеможливлює відновлення оригіналу навіть у разі компрометації бази даних.
- Авторизація повідомлень у WebSocket реалізована на рівні брокера: підписатися на канал може тільки клієнт або спеціаліст, пов'язаний із конкретним зверненням.

Тестування та результати

- Методологія — комбінація ручного функціонального тестування, інтеграційних сценаріїв Postman / Swagger UI й навантажувального тестування Gatling.
- Навантажувальне тестування: 50 одночасних користувачів → середній час авторизації < 10 с, генерації PDF-звіту 0,25–0,5 с.
- За результатами тестування комплекс відповідає функціональним і нефункціональним вимогам; критичних дефектів не виявлено.

Висновки та перспективи розвитку

- Реалізовано повнофункціональний комплекс: клієнтський JavaFX-додаток збирає системні параметри, сервер Spring Boot формує PDF-звіти, а чат WebSocket автоматизує взаємодію клієнта зі спеціалістом .
- Комплекс відповідає всім заявленим функціональним вимогам; під час тестування не виявлено критичних дефектів, інтерфейс підтвердив інтуїтивність для трьох ролей користувачів.
- Архітектура модульна: вона спрощує оновлення компонентів, підвищує захищеність і дозволяє розширювати функціонал без зміни базової.

Подальший розвиток:

- мобільний клієнт і push-сповіщення для швидкої реакції;
- аналітика звернень та дашборди для управлінських рішень;
- механізм підтвердження критичних операцій (видалення).