ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО «ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД

«МІЖРЕГІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ»»
Факультет комп'ютерно-інформаційних технологій
Кафедра комп’ютерних інформаційних систем і технологій

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

Тема «Сенсорна мережа моніторингу надзвичайних ситуацій на виробництві**»**

Виконав студент

Групи ІК-9-22-Б1КНТ

Катеринчик Нікіта Володимирович

Науковий керівник

Дуднік. А.

Допущено до захисту
Завідувач кафедри _____ д.е.н., професор Кавун С.В.
(підпис)

РЕФЕРАТ

Під час виконання даної дипломної роботи було спроєктовано, розроблено, змодельовано прототип системи моніторингу надзвичайних ситуацій на виробництві. його цінність в подальшому використанні для навчальних цілей або як основа для розробки промислових IoT-рішень. Даний прототип системи моніторингу був розроблений з використанням такого стеку технологій: система була розроблена на мові програмування C#, клієнтом виступила платформа Windows Presentation Foundation (WPF) для відображення клієнтського інтерфейсу (UI Дизайну), для відображення даних в режимі реального часу було прийнято рішення використати технологію SignalR, для запису та збереження даних використовувалась база даних SQlite. У результаті поєднання усіх цих технологій вдалося реалізувати програму яка може відстежувати рівень небезпеки на підприємстві, встановлювати рівень загрози і повідомляти про стан підприємства.

Зміст

ВСТУП

На сучасних виробничих підприємствах високий рівень автоматизації та постійно відбуваються складні технологічні процеси через це є велика кількість потенційно небезпечних, загрозливих факторів. На різних підприємствах можуть виникати надзвичайні ситуації, що обумовлені підвищенням температури, задимленням, витоком небезпечних речовин, пожежами або несправністю обладнання. Такі події спричиняють серйозну загрозу для життя та здоров’я працівників, можуть спричинити великі матеріальні збитки для підприємства, а також негативно впливають на навколишнє середовище. Через це особливої актуальності набуває створення ефективних систем моніторингу та швидкого реагування на небезпечні ситуації. Одним із перспективних напрямів забезпечення безпеки на виробництві є використання сенсорних мереж. Сенсорні мережі дозволяють у режимі реального часу здійснювати збір, обробку та передачу інформації про стан виробничого середовища. Завдяки використанню датчиків температури, диму та інших параметрів можна своєчасно виявляти ознаки небезпеки та діяти на випередження щоб уникнути аварійних ситуацій. Використання подібних систем впливає на підвищення рівня безпеки праці, значно зменшує ризики і допомагає покращити контроль над технологічними процесами. Ця тема актуальна оскільки поки є підприємства де можуть трапитись якісь аварійні ситуації буде необхідність у розробці програмного ПО яке буде безперервно вести контроль над станом виробничого середовища, автоматично визначати рівень небезпеки та швидко реагувати і повідомляти користувачів про можливе винекнення загрози, що тільки посилить рівень безпеки для людей і зменшить збитки завдані підприємствам. Використання програмних засобів та сенсорних технологій дає змогу створити ефективну систему контролю, здатну працювати в реальному

часі та забезпечувати швидке реагування на критичні події. Метою цієї дипломної роботи є розробка,

моделювання прототипу сенсорної мережі що буде моніторити надзвичайні ситуації на виробництві, програма буде приймати вхідні дані із сенсорів, визначати рівень небезпеки та відображати інформацію користувачу в зручному інтерфейсі.

Для того щоб досягнути поставленої мети необхідно виконати такі завдання:

1) проаналізувати існуючі системи моніторингу надзвичайних ситуацій;

2) дослідити принципи роботи сенсорних мереж та методи передачі даних;

3) розробити архітектуру програмної системи моніторингу;

4) реалізувати модуль генерації та обробки даних із сенсорів;

5) створити клієнтський застосунок для відображення інформації про стан системи;

6) реалізувати механізм визначення аварійних рівнів небезпеки;

7) провести тестування працездатності створеної системи.

Об’єктом дослідження є процес моніторингу надзвичайних ситуацій на виробничих об’єктах.

Предметом дослідження є методи та програмні засоби побудови сенсорної мережі для виявлення небезпечних ситуацій і передачі даних у режимі реального часу.

У процесі виконання дипломної роботи були використані такі методи дослідження: аналіз наукової та технічної літератури, методи об’єктно-орієнтованого програмування, моделювання роботи сенсорної мережі, тестування програмного забезпечення та аналіз отриманих результатів.

Структура дипломної роботи складається зі вступу, кількох розділів основної частини, висновків, списку використаних джерел та додатків. У першому розділі розглядаються теоретичні аспекти сенсорних мереж та існуючі аналоги систем моніторингу. У другому розділі описується процес проєктування та архітектура програмної системи. Третій розділ присвячений реалізації програмного забезпечення, четвертий тестуванню та аналізу результатів роботи системи.

РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА

1.1 Огляд систем моніторингу безпеки

Системи моніторингу безпеки, також відомі як системи управління інформацією та подіями безпеки (SIEM), це інструменти, що збирають, аналізують та моніторять дані безпеки з різних джерел у мережі організації. Ці системи забезпечують видимість подій у реальному часі, що дозволяє виявляти потенційні загрози, атаки та незвичайні дії.

Системи моніторингу безпеки відіграють ключову роль у забезпеченні безпеки організації. Вони збирають дані з мережевого обладнання, серверів, програмних застосунків і засобів захисту, а потім об'єднують їх воєдино. А отримана інформація аналізується за допомогою складних алгоритмів і правил, що дає змогу виявляти шаблони чи аномалії, які можуть говорити про загрозу.

Для моніторингу й аналізу подій безпеки такі системи використовують різні підходи:

Управління логами - збирання даних із різноманітних джерел (фаєрволів, систем виявлення та запобігання вторгненням, антивірусів), їх обробка та нормалізація для забезпечення послідовного й точного аналізу.

Кореляція та аналіз - зіставлення подій у реальному часі, що допомагає фахівцям із безпеки бачити зв'язки між різними діями та виявляти підозрілу активність, яка може вказувати на порушення безпеки.

Інтеграція з розвідкою загроз - підключення до зовнішніх джерел інформації про відомі загрози та вразливості, що дає можливість порівнювати мережеву активність із базою відомих шкідливих індикаторів і підвищує точність виявлення.

Генерація сповіщень і звітів - створення попереджень на основі заданих правил і встановлених значень, які надсилаються відповідальним за безпеку працівникам для швидкого реагування. Звіти дозволяють оцінити загальний стан безпеки організації та дотримуватися вимог регуляторів стосовно захисту даних і приватності. Упровадження та підтримка надійних систем моніторингу безпеки має вирішальне значення для захисту мережі та конфіденційних даних організації.

Ось кілька рекомендацій для забезпечення ефективного моніторингу: Комплексне покриття, системи моніторингу мають охоплювати всі критичні ділянки мережі та IT-інфраструктури, такі як мережевий трафік, системні логи, логи застосунків і активність користувачів. Аналіз даних із різних джерел дає організації повноцінну картину її рівня безпеки.

Регулярні оновлення, системи моніторингу потрібно постійно перевіряти й оновлювати, щоб вони фіксували актуальні події. Це стосується правил,

алгоритмів кореляції та каналів розвідки загроз. Своєчасне оновлення дозволяє системі реагувати на нові загрози та зміни у вимогах безпеки. Навчання персоналу, співробітники відділу безпеки мають пройти спеціальну підготовку, аби правильно інтерпретувати сповіщення та реагувати на них. Навчання повинно охоплювати розуміння типів подій безпеки, розслідування тривог і дотримання процедур реагування на інциденти. Кваліфікований персонал здатен знизити потенційні ризики та швидко реагувати на загрози. Інтеграція з процесами реагування на інциденти, системи моніторингу мають бути тісно пов'язані з процедурами реагування на інциденти. При виявленні потенційної загрози система автоматично запускає відповідний робочий процес, що

забезпечує швидку та узгоджену реакцію. Така інтеграція скорочує час виявлення й нейтралізації загроз, мінімізуючи їхній вплив на організацію. Дотримання цих рекомендацій дозволяє організаціям підвищити ефективність моніторингу безпеки та надійніше захищати свої мережі й конфіденційну інформацію.

1.2 Сенсорні мережі (IoT концепція)

Internet of Things (IoT) - це концепція, за якою різноманітні фізичні пристрої поєднуються через інтернет, отримуючи змогу взаємодіяти та обмінюватися даними. Основна її ідея полягає в тому, щоб наділити об'єкти (інтелектом) і зв'язати їх між собою, створивши єдиний простір для збору й обробки інформації. Це дає змогу покращити якість життя, оптимізувати процеси та підвищити ефективність. Отже, інтернет речей - це пристрої, об'єднані бездротовою мережею для виконання конкретних завдань.

Історія та суть поняття

Термін уперше з'явився 1999 року: технолог Кевін Ештон у своїй презентації для Procter & Gamble описав концепцію збору даних у реальному часі через підключення до мережі й дав їй цю назву. Пік розвитку припав на 2008–2009 роки, саме в цей час кількість підключених до інтернету пристроїв перевищила чисельність населення Землі. Відтоді поняття інтернету речей набуло нового значення й стало невід'ємною частиною життя сучасної людини. Цьому сприяли три ключові фактори: розвиток бездротових технологій, зменшення розмірів сенсорів, зростання обчислювальної потужності. Гарний приклад - розумні годинники. Компактні й зручні, вони є частиною екосистем Apple або Android. Такі пристрої постійно відстежують показники здоров'я та фізичної активності власника і, головне, передають ці дані на смартфон для подальшого аналізу.

Зрозуміти, що таке інтернет речей із чого він складається, нам допоможе огляд ключових компонентів, кожен з яких відіграє важливу роль у його функціонуванні: пристрої - сенсори для збору даних із навколишнього середовища: температура, вологість, світло, рух, а також пристрої, що виконують дії на основі отриманих даних: увімкнення світла або регулювання температури, зв’язок - мережеві протоколи та інші технології для обміну відомостями між основними компонентами та центральними системами, обробка інформації - хмарні платформи, що забезпечують зберігання й аналіз даних у реальному часі, а також аналітичні інструменти для виявлення закономірностей і прогнозування, інтерфейси - для користувача інтернет речей IoT це мобільні та веб застосунки, що надають доступ до даних, управління пристроями та налаштування параметрів, а також програмні інтерфейси для взаємодії між різними системами та додатками, безпека - шифрування інформації для захисту від несанкціонованого доступу, а також механізми, що обмежують неавторизованих користувачів.

Що таке мережа IoT - це концепція, яка дала змогу зробити великий крок у галузі технологій. Вона змінила сам спосіб взаємодії людини з навколишнім світом. Це підвищило продуктивність та енергоефективність різних секторів, а головне, автоматизувало більшість процесів, зменшуючи обсяги ручної праці. Основний принцип IoT полягає в забезпеченні зручної взаємодії пристроїв між собою, а також людини з пристроями.

Базові принципи функціонування інтернету речей такі: по-перше, об'єкти під'єднуються до мережі - різноманітні пристрої отримують унікальні мережеві адреси та стають частиною єдиного мережевого простору. По-друге, відбувається обмін даними - складові системи можуть як отримувати інформацію з мережі, так і передавати власні дані іншим пристроям і системам. По-третє, забезпечується автоматизована робота - об'єкти функціонують без участі людини, виконуючи завдання за заданими програмами.

Спрощено схема роботи IoT виглядає так: сенсори та пристрої фіксують ключові параметри, передають їх по ланцюжку до центральних систем, де аналітичні інструменти та програми формують графіки, звіти та прогнозні моделі. Це дозволяє не лише реагувати на поточні події, а й передбачати їх, запобігаючи потенційним проблемам. Ця технологія є актуальною як у повсякденному житті, так і в багатьох інших сферах, наприклад в промисловості, транспорті, енергетиці, роздрібній торгівлі, логістиці та складському господарстві, освіті, фінансових послугах, будівництві, готельному бізнесі й телекомунікаціях. Розумні пристрої вже стали невід'ємною частиною повсякдення, а використання інтернету речей у різних галузях економіки здатне значно спростити й оптимізувати безліч процесів, мінімізуючи витрати та зменшуючи обсяг зусиль, необхідних для досягнення поставлених цілей.

1.3 Клієнт-серверна архітектура

Клієнт - це комп'ютер на стороні користувача, який надсилає запит до сервера для отримання інформації або виконання певних дій.

Сервер, у свою чергу, - це потужніший комп'ютер або обладнання, призначене для виконання програмного коду, сервісних функцій за запитами клієнтів, надання доступу до ресурсів, а також зберігання інформації та баз даних.

Модель такої системи працює так: клієнт відправляє запит на сервер, де він обробляється, після чого готовий результат повертається клієнтові. Сервер здатен обслуговувати кількох клієнтів одночасно. Якщо надходить більше одного запиту, вони стають у чергу й виконуються послідовно. Деякі запити можуть мати пріоритет - такі виконуються раніше. Функції які виконує сервер: зберігання, доступ, захист і резервне копіювання даних, обробка клієнтських запитів, надсилання відповіді клієнту. Функції клієнта: надання користувацького інтерфейсу, формулювання запиту до сервера та його відправка, отримання результатів і надсилання додаткових команд (на

додавання, оновлення або видалення даних). Клієнт-серверна архітектура визначає принципи взаємодії між комп'ютерами, а правила цієї взаємодії описуються протоколом.

Мережевий протокол - це набір певних правил, за якими комп'ютери обмінюються даними в мережі.

Основні мережеві протоколи:

1) TCP/IP - стек протоколів передачі даних, що позначає всю мережу, яка працює на основі двох протоколів: TCP і IP.

2) TCP (Transfer Control Protocol) - встановлює надійне з'єднання між пристроями, передає інформацію та підтверджує її отримання.

3) IP (Internet Protocol) - відповідає за доставку повідомлень за правильною адресою; дані при цьому розбиваються на пакети, які можуть доставлятися різними шляхами.

4) MAC (Media Access Control) - протокол ідентифікації мережевих пристроїв; кожен пристрій, підключений до інтернету, має унікальну MAC-адресу.

5) ICMP (Internet Control Message Protocol) - відповідає за обмін службовою інформацією, але не використовується для передачі даних.

6) UDP (User Datagram Protocol) - керує передачею даних без перевірки отримання, що забезпечує вищу швидкість порівняно з TCP.

7) HTTP (Hyper Text Transfer Protocol) - протокол передачі гіпертексту, на якому працюють усі вебсайти; він запитує дані (сторінки, файли) у віддаленої системи.

8) FTP (File Transfer Protocol) - протокол передачі файлів із файлового сервера на комп'ютер користувача.

9) SSH (Secure Shell) - протокол для віддаленого керування системою через захищений канал.

10) POP3 (Post Office Protocol) - стандартний поштовий протокол, який відповідає за доставку пошти.

Існує два основні типи клієнт-серверної архітектури: дворівнева та трирівнева.

Дворівнева архітектура складається з двох вузлів: сервера та клієнта. Сервер отримує запити й надсилає відповіді, використовуючи лише власні ресурси, без допомоги сторонніх систем. Клієнт же зі свого боку, забезпечує користувацький

інтерфейс. Принцип роботи простий: сервер приймає запит, обробляє його і одразу повертає результат клієнтові.

Трирівнева архітектура складається з трьох компонентів: рівня представлення даних (користувацький інтерфейс), прикладного компонента (сервер додатків) і рівня керування ресурсами (сервер бази даних, який надає інформацію). У цій моделі запит клієнта обробляється кількома серверами, що дозволяє розподілити навантаження та підвищити ефективність роботи системи.

Трирівневу архітектуру можна розширити до багаторівневої (N-tier, Multi-tier) шляхом додавання додаткових серверів. Така архітектура дає змогу підвищити ефективність роботи інформаційної системи та оптимізувати розподіл її програмно-апаратних ресурсів.

1.4 Real-time системи (SignalR, WebSocket)

WebSocket - це протокол, який забезпечує повнодуплексний зв'язок через єдине TCP-з'єднання. Він робить можливою двосторонню взаємодію між клієнтом і сервером, дозволяючи асинхронно надсилати та отримувати дані без зайвих витрат, які характерні для HTTP-опитування.

У чому переваги WebSocket? це зв'язок у реальному часі: протокол забезпечує миттєву передачу даних, що робить його ідеальним для застосунків, які потребують негайних оновлень, чатів, інформаційних панелей у реальному часі та онлайн-ігор. Ефективність: на відміну від традиційних методів HTTP-опитування або довгого опитування (long polling), WebSocket знижує накладні витрати завдяки підтримці постійного з'єднання, мінімізуючи затримку й зменшуючи мережевий трафік. Повнодуплексний зв'язок: WebSocket підтримує одночасну передачу даних в обох напрямках, дозволяючи клієнту й серверу надсилати та отримувати повідомлення незалежно одне від одного. І

нарешті, масштабованість: протокол ефективно обробляє велику кількість одночасних з'єднань, що робить його придатним для масштабованих високопродуктивних систем.

Але WebSocket має й недоліки. Реалізація та управління з'єднаннями WebSocket є складнішими порівняно з традиційною HTTP-комунікацією. Хоча сучасні браузери підтримують протокол, старі версії можуть його не підтримувати, що потребує використання резервних механізмів або альтернативних підходів. Також можливі проблеми з обмежувальними брандмауерами чи проксі-серверами, які блокують трафік WebSocket.

SignalR - це високорівнева бібліотека, побудована на основі WebSocket (та інших транспортних механізмів), яка значно спрощує роботу з вебзастосунками реального часу в екосистемі .NET. Вона абстрагує складність управління з'єднаннями та надає простий API для трансляції повідомлень клієнтам і обробки взаємодії між клієнтом і сервером.

У чому переваги SignalR? це спрощення розробки: бібліотека бере на себе всю складність управління з'єднаннями WebSocket, маршрутизацію повідомлень і обробку помилок, дозволяючи розробникам зосередитися на логіці застосунку. Крос-платформова підтримка: SignalR працює як із серверними застосунками .NET, так і з клієнтськими JavaScript-фреймворками, що робить його придатним для вебзастосунків, настільних і мобільних

застосунків. Масштабованість: SignalR підтримує велику кількість одночасних підключень і надає можливості масштабування на кілька серверів, зокрема через хмарні рішення, як-от Azure SignalR Service. І нарешті, механізми резервного перемикання: бібліотека автоматично переходить на альтернативні транспортні механізми - Server-Sent Events (SSE) або long polling - для клієнтів, які не підтримують WebSocket, забезпечуючи широку сумісність із різними браузерами та пристроями.

Але й тут не обійшлося без мінусів адже попри численні переваги, SignalR має й певні обмеження. По-перше, бібліотека тісно пов'язана з екосистемою .NET, що робить її менш придатною для застосунків, створених на інших технологіях. По-друге, хоча SignalR спрощує розробку, він може створювати певні накладні витрати на продуктивність порівняно з чистими реалізаціями WebSocket. Крім того, попри те, що SignalR абстрагує багато складнощів обміну даними в реальному часі, у складних сценаріях все одно може знадобитися додаткове налаштування та усунення несправностей.

Передача даних у режимі реального часу необхідна для систем, у яких важливе миттєве оновлення інформації. До таких систем належать чати, де повідомлення повинні надсилатися без затримок, панелі моніторингу та інші інструменти спостереження за станом процесів у реальному часі, а також сервіси спільного редагування документів, що потребують синхронізації змін між користувачами. Крім того, цей підхід широко використовується в онлайн-ігрових платформах, додатках для відстеження котирувань на фондовому ринку та системах, які відображають результати спортивних подій у режимі реального часу.

1.5 Бази даних у системах моніторингу

Сучасний світ генерує величезні обсяги інформації щосекунди. Дані про покупки в інтернет-магазинах, банківські операції, медичні записи, робочі документи та хмарні сервіси - усе це зберігається та обробляється за допомогою баз даних. Саме вони є ключовим інструментом для організації, структурування та

ефективного управління інформацією. У найпростішому розумінні база даних - це впорядковане електронне сховище, яке дозволяє зручно зберігати великі обсяги інформації та швидко отримувати до неї доступ. Наприклад, елементарною формою бази даних можна вважати таблицю клієнтів або список замовлень. Проте сучасні бази даних значно складніші та функціональніші, ніж

звичайні таблиці чи документи. Вони складаються з кількох основних компонентів. Першим є дані, тобто безпосередня інформація, яка зберігається в системі: імена користувачів, дати, числові значення, описи та інші параметри.

Другим компонентом є схема бази даних, яка визначає структуру зберігання інформації, типи даних, а також зв’язки між різними таблицями або сутностями. Саме схема забезпечує логічну організацію всієї інформації. Третім і найважливішим елементом є система керування базами даних (СКБД абоDBMS). Це програмне забезпечення, яке виступає посередником між користувачем або застосунком і самою базою даних. Воно відповідає за виконання запитів, забезпечення безпеки даних, підтримку їхньої цілісності, оптимізацію швидкості доступу та адміністрування інформації. Таким чином, база даних - це не просто сховище інформації, а складна система, що забезпечує надійне, структуроване та ефективне управління даними у сучасних програмних рішеннях.

Центральним компонентом будь-якої бази даних є система керування базами даних (СКБД, DBMS - Database Management System). Саме вона виступає основним програмним посередником між користувачем або прикладним застосунком і електронним сховищем інформації, забезпечуючи всі операції з даними. Коли користувач надсилає запит до бази даних, СКБД бере на себе його обробку та виконує низку ключових завдань. Перш за все, вона приймає запити від прикладної частини системи, які зазвичай формулюються за допомогою стандартної мови SQL. Наприклад, це може бути запит на отримання всіх замовлень конкретного клієнта. Далі система аналізує запит, знаходить необхідні дані на фізичному носії, перевіряє права доступу користувача та виконує

відповідну операцію зчитування або модифікації інформації. Окрім цього, СКБД забезпечує коректність обробки даних і гарантує їхню цілісність.

Важливою функцією системи є також підтримка одночасної роботи багатьох користувачів. Вона запобігає конфліктам при паралельному доступі до одних і тих самих даних, забезпечуючи узгодженість і стабільність інформації в базі. Ключова відмінність бази даних від звичайного файлу полягає в її чіткій структурі. Дані зберігаються не хаотично, а відповідно до спеціально визначеної схеми (Database Schema), яка задає правила їх організації та взаємозв'язків. У реляційних базах даних інформація зберігається у вигляді електронних таблиць, що складаються зі стовпців і рядків. Стовпці визначають тип даних, які зберігаються, наприклад "Ім'я", "Ціна" або "Дата", а рядки містять окремі записи, наприклад інформацію про конкретного клієнта.

Таблиці можуть бути пов'язані між собою за допомогою ключів: наприклад, таблиця "Клієнти" пов'язується з таблицею "Замовлення" через ідентифікатор клієнта, що забезпечує цілісність і зв'язність даних.

Робота з базою даних базується на чотирьох основних операціях, які об'єднують під акронімом CRUD: Create - створення нових записів, Read - читання або отримання даних, Update - оновлення наявної інформації та Delete - видалення записів. Коли програма виконує одну з цих операцій, вона надсилає запит до системи керування базами даних (СКБД). СКБД обробляє запит, перевіряє дотримання правил цілісності, наприклад чи існує клієнт для створення замовлення, і лише після цього вносить зміни до фізичного сховища. Завдяки цьому дані залишаються впорядкованими, логічно пов'язаними та доступними для подальшого використання.

Бази даних забезпечують низку важливих переваг для сучасних інформаційних систем, бізнес-платформ та програмних продуктів. Однією з головних переваг є швидкий доступ до інформації: бази даних дають змогу отримувати необхідні

дані за частки секунди, що особливо важливо для систем, які працюють у режимі реального часу, зокрема онлайн-магазинів, банківських сервісів чи мобільних застосунків. Завдяки структурованому зберіганню інформації у таблицях із чітко визначеними зв’язками та правилами інтеграції забезпечується впорядкованість даних, зменшується кількість дублювань і підвищується точність роботи з інформацією.

Сучасні бази даних також вирізняються високою масштабованістю та надійним рівнем безпеки. Вони здатні ефективно працювати навіть при значному збільшенні обсягів даних і навантаження на систему, не втрачаючи продуктивності. Для захисту інформації використовуються механізми шифрування, контролю доступу та резервного копіювання. Крім того, бази даних
відіграють важливу роль у проведенні аналітики та підтримці процесу прийняття рішень, оскільки дозволяють швидко обробляти великі обсяги інформації та отримувати цінні висновки для бізнесу або наукових досліджень. Ще однією важливою перевагою є можливість автоматизації та інтеграції різних систем, що спрощує обмін даними, взаємодію між програмними компонентами та створення комплексних цифрових рішень.

На початкових етапах розвитку IT-індустрії процес керування базами даних був досить складним навіть для професійних програмістів, оскільки базувався на застарілих ієрархічних та мережевих моделях організації інформації. Значний прорив у цій сфері відбувся у 70-х роках XX століття завдяки програмісту компанії IBM Едгар Кодд, який запропонував реляційну модель зберігання та обробки даних. Саме вона стала основою для більшості сучасних систем керування базами даних. На сьогодні бази даних умовно поділяють на дві основні категорії: традиційні реляційні бази даних SQL та сучасні нереляційні бази даних NoSQL.

1.6 Висновки до розділу

У першому розділі було розглянуто основні теоретичні аспекти побудови сучасних систем моніторингу безпеки та технології, що використовуються для їх реалізації. Проведений аналіз показав, що системи моніторингу є важливим елементом забезпечення безпеки об’єктів, оскільки дозволяють здійснювати постійний контроль стану середовища, оперативно виявляти загрози та швидко реагувати на надзвичайні ситуації.

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Загальна архітектура системи

Розроблена система моніторингу надзвичайних ситуацій має призначення збору, передачі, обробки та візуалізації телеметричних даних із сенсорних вузлів у режимі реального часу. Архітектура системи була побудована за клієнт-серверним принципом і складається з таких основних модулів як:

1) модуль емуляції сенсорів, який відповідає за генерацію даних;

2) серверна частина, яка відповідає за обробку та збереження даних;

3) клієнтський WPF-застосунок, який використовується для відображення (UI Дизайну).

Основною задачею системи є оперативне отримання параметрів навколишнього середовища та визначення можливих аварійних ситуацій. Для забезпечення передачі даних у режимі реального часу використовується технологія SignalR, яка забезпечує двосторонній зв’язок між клієнтами та сервером.

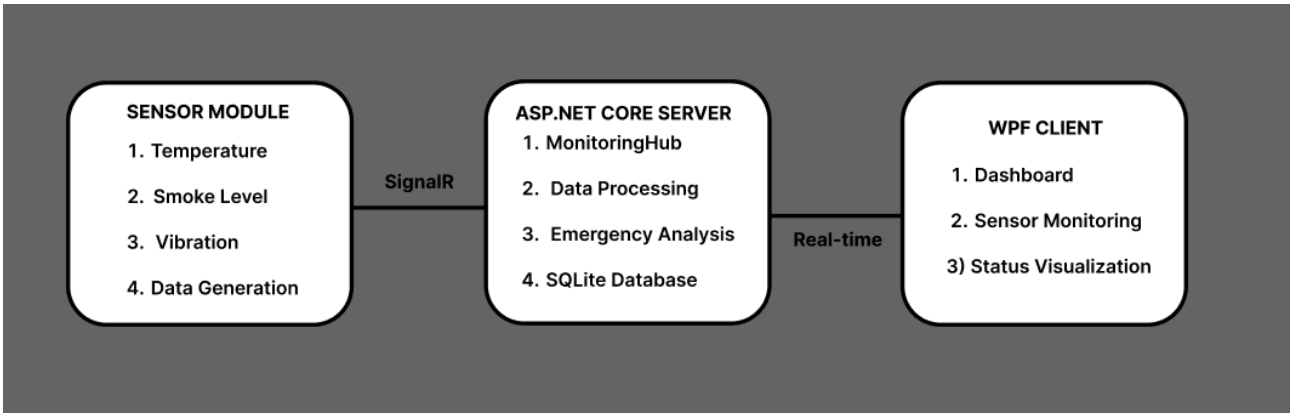


Рис. 2.1 – схема архітектури

2.2 Проєктування сенсорного модуля

Оскільки справжній датчик було б проблемно реалізувати через сукупність багатьох факторів було прийнято рішення використати окремий консольний додаток на C# для імітації датчиків. Основною функцією цього модуля є імітація роботи фізичних сенсорних пристроїв та передача телеметричних даних на сервер. Тож у системі було реалізовано декілька віртуальних сенсорів, які носять назву: SENSOR-1, SENSOR-2, SENSOR-3, SENSOR-4, SENSOR-5. Кожен з цих імітованих сенсорів закріплений за окремою робочою зоною, кількість зон дорівнює кількості датчиків що означає на кожну зону по датчику, ось перелік доступних зон на нашому імітованому підприємстві: Склад, Виробництво, Сховище хімікатів, Котельна, Серверна. Сенсорний модуль генерує наступні параметри: рівень температури, рівень диму, рівень вібрації, і час коли відбулась аварія. Для генерації даних ми використаємо клас на мові програмування C# під назвою Random який дає можливість нам створювати випадкові значення у визначених діапазонах. Передача даних на сервер

здійснюється через SignalR-з’єднання параметри виставлені так щоб дані оновлювались кожні 10 секунд. Формування ж самих даних виконується у вигляді класа SensorData.

20

Табл 2.1 – робота сенсорного модуля

Назва сенсора	Робоча зона	Параметри, що генеруються	Інтервал оновлення	Технологія передачі
SENSOR-1	Склад	Температура, дим, вібрації, час аварії	10 секунд	SignalR
SENSOR-2	Виробництво	Температура, дим, вібрації, час аварії	10 секунд	SignalR
SENSOR-3	Сховище хімікатів	Температура, дим, вібрації, час аварії	10 секунд	SignalR

2.3 Серверна частина системи

Серверна частина системи була реалізована на платформі ASP.NET Core. До функцій які виконує сервер належать: прийом даних з датчиків, обробка інформації яку прийняв сервер, визначення рівня небезпеки, зберігання отриманих даних в базу даних SQlite, та передача даних клієнтам через wpf. Основним компонентом серверної частини є MonitoringHub, реалізований за допомогою SignalR. Після того як дані передаються сенсорами, сервер приймає телеметричний пакет, аналізує параметри, зберігає дані у базі даних, надсилає оновлені дані до всіх підключених клієнтів. Для роботи з базою даних

використовується Entity Framework Core. Як систему збереження даних було обрано базу даних SQlite через просту інтеграцію, малий розмір, відсутня необхдність окремого серверу бази даних, висока швидкість роботи для локальних систем моніторингу.

21

Табл 2.2 – функції серверної системи

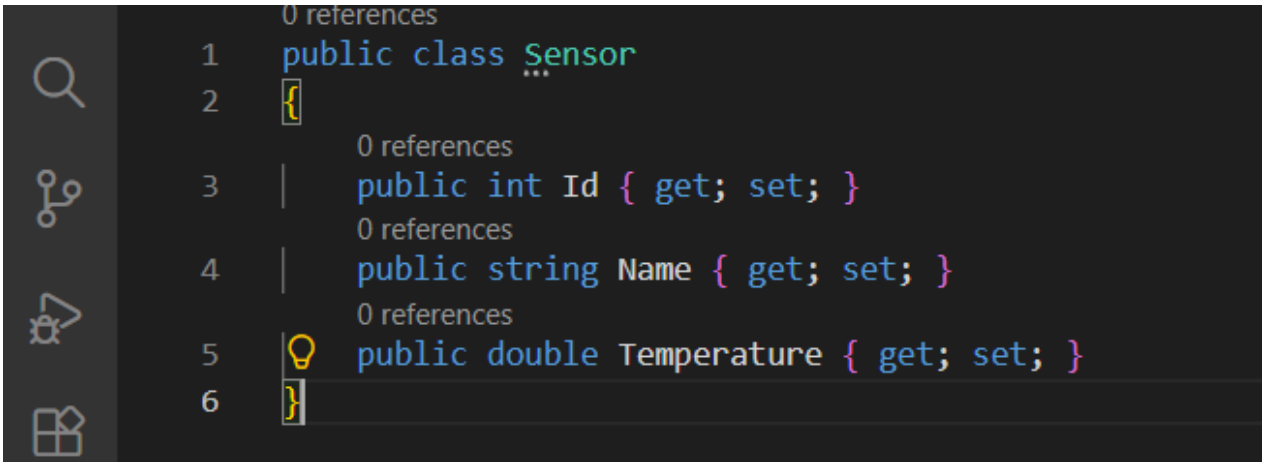
Компонент	Призначення	Використані технології
ASP.NET Core Server	Основна серверна логіка системи	ASP.NET Core
MonitoringHub	Прийом та передача даних у реальному часі	SignalR
Обробка телеметрії	Аналіз параметрів та визначення рівня небезпеки	C#
Модуль збереження даних	Запис телеметрії у базу даних	Entity Framework Core
База даних	Збереження історії даних та аварій	SQLite
WPF-клієнт	Отримання оновлень та відображення інформації	WPF + SignalR

2.4 Модель даних

У мові програмування C# модель - це клас, який представляє якусь річ з реального світу, бізнес-дані або структуру бази даних. Вона містить лише властивості (дані) та найпростішу логіку, виступаючи контейнером для зберігання та передачі інформації між різними частинами програми.

Найчастіше моделі застосовуються у архітектурі застосунків типу MVC (Model-View-Controller), WPF, ASP.NET та інших .NET-проєктах. Стосовно моделей, їх у проєкті є декілька і кожна з них відіграє свою важливу роль. Моделі є важливою частиною об’єктно-орієнтованого програмування, оскільки вони дозволяють створювати зрозумілу архітектуру застосунку, спрощують підтримку коду та забезпечують повторне використання компонентів.

Ось простий приклад у якому модель Sensor описує сенсор моніторингу та містить його ідентифікатор, назву і температуру. Завдяки моделям дані в програмі стають структурованими та зручними для передачі між різними компонентами системи, наприклад між базою даних, сервером і користувацьким інтерфейсом.



кадр. 2.2 – приклад моделі даних

2.5 Клієнтський застосунок (WPF)

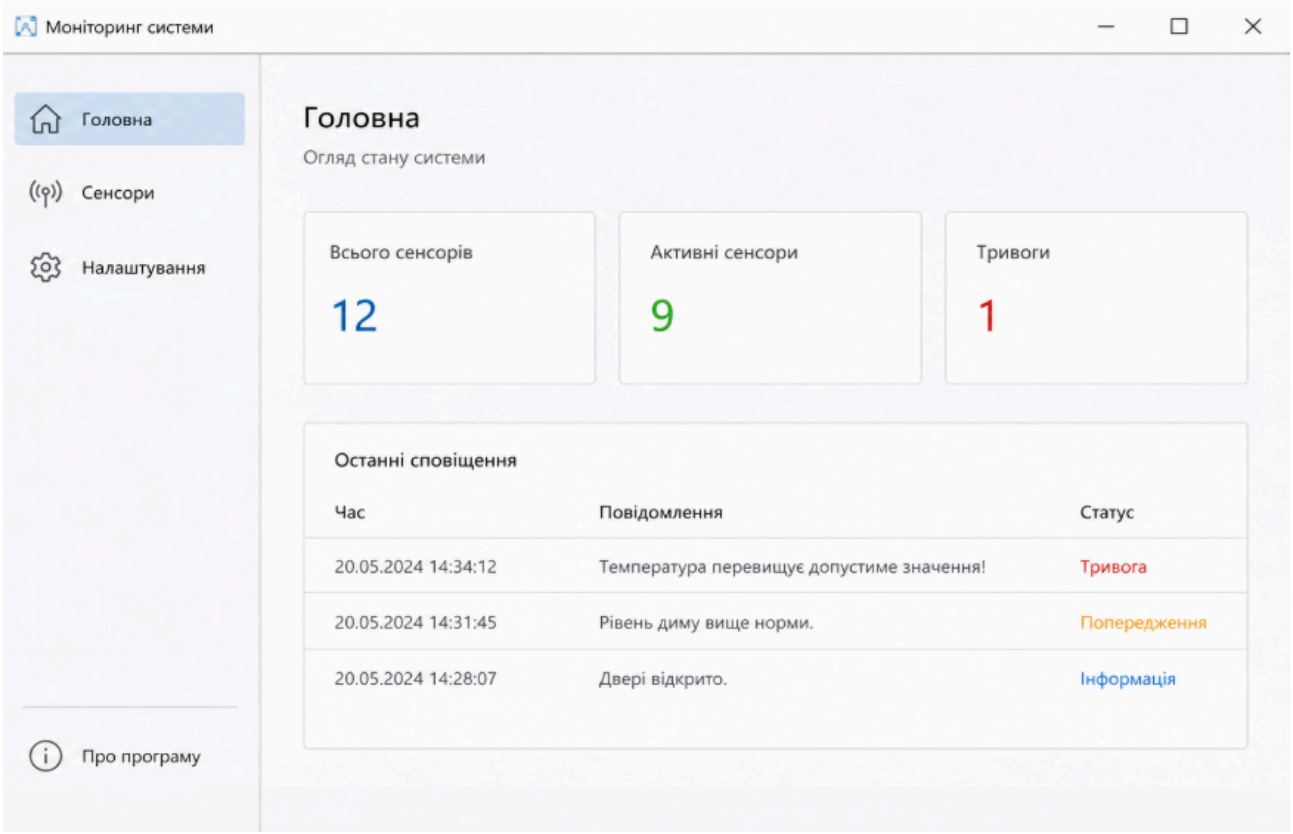
Клієнтський застосунок WPF (Windows Presentation Foundation) — це настільна програма для операційної системи Windows, створена на платформі .NET із

використанням технології WPF. Основним призначенням цієї програми є створення умов для забезпечення взаємодії користувача із системою через графічний інтерфейс. WPF дозволяє створювати сучасні віконні програми з кнопками, таблицями, анімаціями, стилями та іншими елементами інтерфейсу. Особливістю WPF є використання мови розмітки XAML (eXtensible Application Markup Language), яка застосовується для опису зовнішнього вигляду програми. Логіка роботи застосунку при цьому реалізується мовою C#. Такий підхід дозволяє розділити інтерфейс і програмний код, що спрощує розробку та підтримку проєкту.

Клієнтський WPF-застосунок може отримувати дані із сервера, бази даних або API, обробляти їх та відображати користувачу у зручному вигляді. Наприклад, у системах моніторингу безпеки WPF-клієнт може показувати показники сенсорів, стан обладнання, сповіщення про небезпеку та статистику в режимі реального часу.

Однією з важливих переваг WPF є підтримка шаблону MVVM (Model-View-ViewModel), який допомагає створювати структурований та масштабований код. Окрім того, WPF підтримує прив’язку даних (Data Binding), стилізацію інтерфейсу та роботу з мультимедійними елементами.

Ось приклад вигляду wpf застосунку:



кадр. 2.1 – wpf застосунок

2.6 Алгоритм визначення аварійних станів

Алгоритм визначення аврійних станів відіграє дуже важливу роль в цьому проєкті, адже саме він буде перевіряти чи не перевищують значення показників, значення допустимих параметрів і тим самим говорити нам про стан підприємства. Якщо значення показників будуть перевищувати норму, в такому разі в залежності від перевищення норми показників ми будемо отримувати одне з повідомлень що буде свідчити про рівень загрози у відповідному цеху підприємства наприклад Normal - нормальний стан загрози немає, Warning – попередження може статись аварія, Critical – сталася аварія. Такий алгоритм можна реалізувати багатьма способами у цьому розділі я преведу найпростіший з них.

Приклад:

```
if (data.Temperature > 80)
            {
                status = "Критично";
                color = Brushes.Red;
            }
            else if (data.Temperature > 60)
            {
                status = "Загроза";
                color = Brushes.Orange;
            }
```

кадр. 2.4 – фрагмент коду визначення рівня аварійності

2.7 Взаємодія компонентів системи

Цей підрозділ присвячений не великому роз’ясненню того як усі компоненти зібраної системи взаємодіють між собою. Оскільки для побудови цієї системи я використовував Клієнт-Серверну архітектуру зрозуміло що емулятор сенсорів буде генерувати інформацію передавати її на сервер де вона буде у подальшому зберігатись і коли клієнт буде робити запит до серверу він буде отримувати результат. Ось як це має працювати у спрощеному вигляді: отже спочатку ми запускаємо сервер, потім запускаємо wpf застосунок, і на остачу емулятор сенсорів. Після запуску емулятор сенсорів починає генерувати випадкові дані, генерація даних залежить від встановленого часу генерації даних, і верхнього

ліміту встановлених значень у програмі. Наступним кроком сгенеровані дані приймає сервер, який займається обробкою даних, він зберігає дані у базі даних SQlite і перевіряє дані на визначення небезпеки, а потім wpf застосунок відображає оновлені дані. У третьому розділі вже будемо обговорювати повну роботу застосунку.

2.8 Висновки до розділу

У цьому розділі ми почали розглядати з чого складається і як побудована уся система. Ми розглянули архітектуру системи, та з яких компонентів вона складається, що до неї належить. Це такі компоненти як: модуль генерації сенсорів, серверна частина програми, а також wpf застосунок для відображення дизайну. І зрозуміли що кожен з цих компонентів окремо має ще свої власні компоненти, які усі разом допомагають зібрати все до купи та змусити працювати.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Середовище розробки

Цей розділ цілком грунтується на коді написаної програми. В ньому буде описано практичне використання коду. У даному розділі будуть часто використовуватись англійські слова з коду для спрощеного пояснення його роботи. Отже уся програма програма була написана у (IDE) Visual Studio, ця програма була обрана незважаючи на її громосткість та складність, через те що вона містить досить зручний редактор коду для написання програм на мові C#. Та має велику кількість бонусів, у Visual Studio можна встановити багато

зручних плагінів для полегшення роботи, також там присутня функція авто доповнення коду, надання підказок, перевірка помилок у реальному часі. А для реалізації теми даної дипломної роботи програма досить добре підходить.

Ось дерево папок з яких складається проєкт:

Табл. 3.1 – Структура папок у розробленому проєкті

ПАПКИ	ВМІСТ
EmergencyMonitoring	Файл sln
EmergencyMonitoring.Server	Файли Сервера
EmergencyMonitoring.Sensor	Файли Сенсорів
EmergencyMonitoring.WPF	Файли WPF
EmergencyMonitoring.Shared	Файли спільних моделей

3.2 Реалізація сенсорного модуля

У сенсорному модулі окрім базових файлів (bin, obj) та файлу конфігурації EmergencyMonitoring.Sensor.csproj який містить в собі деякі налаштування є один головний файл Program.cs у якому прописаний весь головний код цього модуля.

Program.cs

Реалізує консольний симулятор IoT-сенсорів для системи аварійного моніторингу, який працює через технологію SignalR та передає дані на сервер у режимі реального часу. На перших строках програми підключаються необхідні

бібліотеки для роботи зі спільними моделями даних та бібліотекою SignalR. Далі створюється підключення до SignalR Hub за адресою monitoringHub, у випадку втрати зв’язку вмикається автоматичне перепід’єднання та запускається з’єднання із сервером.

Після чого створюється масив сенсорів, у якому для кожного датчика задається унікальний ідентифікатор, назва зони за яку цей датчик відповідальний та окремий колір для відображення інформації в консолі. Що дає змогу імітувати роботу кількох незалежних один від одного пристроїв моніторингу в різних частинах підприємства, наприклад на складі, у серверній або хімічному сховищі.

Для коректної роботи паралельних потоків створюється об’єкт блокування consoleLock, який захистить нас від одночасного запису декількох сенсорів у консоль. Також окремо реалізований метод GenerateValues, який займається генеруванням випадкових показників температури, рівня диму та вібрації у заданих діапазонах. Цей метод використовується для симуляції реальної роботи IoT-сенсорів та постійного оновлення даних.

Далі створюється CancellationTokenSource, який дозволяє коректно завершити виконання всіх фонових задач при зупинці програми. Основна логіка реалізована через набір паралельних асинхронних задач sensorTasks. Для кожного сенсора запускається окремий цикл, який постійно:

Займається генерацією нових показників, створює об’єкт SensorData, відправляє дані на сервер завдяки SendAsync, і виводить інформацію в консоль у відповідному для цієї інформації кольорі.

Після запуску додатку програма відкриває консольний інтерфейс: очищає екран, виводить заголовок системи, кількість сенсорів які активні, інтервал за який оновлюються дані та відповідний колір для кожного датчика. Що забезпечує зручне відображення інформації під час роботи симулятора.

При завершенні програма постійно очікує натискання клавіші 0. Після натискання цієї клавиші відбувається скасування всіх асинхронних задач через CancellationToken, очікується завершення роботи всіх сенсорів і програма правильно завершує свою роботу. Таким чином, даний код реалізує повноцінну систему симуляції датчиків аварійного моніторингу з передачею даних у реальному часі між консольним клієнтом і сервером ASP.NET Core.

3.3 Реалізація серверної частини

Program.cs

Серверна частина програми охоплює більше папок, та код в цій частині програми налаштовує серверну частину системи аварійного моніторингу на базі ASP.NET Core, для роботи з REST API, бібліотекою SignalR та базою даних SQLite.

Спочатку підключаються необхідні для роботи простори імен: контекст бази даних (Data), SignalR-хаб (Hubs), сервіси бізнес-логіки (Services) та Entity Framework Core для роботи з базою даних. Це дозволяє серверу обробляти запити, зберігати дані та передавати їх у реальному часі клієнтам.

Наступним кроком створюється об’єкт WebApplicationBuilder, який відповідає за налаштування всього веб-застосунку. Через нього реєструються сервіси в контейнері залежностей.

У секції налаштування сервісів додаються контролери (AddControllers()), що дає нам можливість використовувати REST API для роботи з HTTP-запитами,

а також SignalR (AddSignalR()), який забезпечує зв’язок між сервером і клієнтами у реальному часі .

Після цього налаштовується база даних через Entity Framework Core. Реєструється AppDbContext, який підключається до SQLite бази даних emergency.db. Що говорить про те що всі дані системи моніторингу (наприклад, показники сенсорів або тривоги) можуть зберігатися локально у файлі бази.

Далі додається сервіс AlertService із життєвим циклом Scoped, тобто він створюється один раз на кожен HTTP-запит або SignalR-операцію. Цей сервіс, відповідає за обробку аварійних ситуацій і генерацію сповіщень.

Далі йде налаштування CORS (Cross-Origin Resource Sharing), він дозволяє клієнтам з будь-яких джерел підключатися до сервера. У блоці політики дозволяються всі заголовки, методи та робота з даними, а також знімається обмеження на домен, що відкриває максимальну доступність для клієнтських додатків (наприклад WPF або веб-клієнтів).

Після конфігурації сервісів викликається builder.Build(), що створює готовий екземпляр веб-застосунку.

Далі налаштовується middleware-пайплайн: активується CORS (UseCors()), підключаються контролери (MapControllers()), а також реєструється SignalR-хаб MonitoringHub за маршрутом /monitoringHub. Саме через цей endpoint клієнти підключаються для отримання даних у реальному часі.

На прикінці викликається app.Run(), який запускає сервер і переводить його в режим прослуховування HTTP-запитів. Таким чином, цей код формує повноцінний backend для системи моніторингу, що підтримує API, відображення інформації у реальному часі обмін даними та роботу з базою даних.

Controllers

SensorController.cs

Задача цього коду полягає у реалізації API-контролера у серверній частині системи аварійного моніторингу, він відповідає за отримання даних з бази даних для конкретного сенсора.

Перш за все підєднуються необхідні для роботи простори імен: Data для доступу до контексту бази даних, Controllers для створення API-контролера та EntityFrameworkCore для виконання асинхронних запитів до бази даних.

Далі оголошується простір імен EmergencyMonitoring.Server.Controllers, у якому знаходиться сам контролер SensorController. Атрибут [ApiController] вказує, що це Web API контролер із автоматичною обробкою запитів і валідацією, а [Route("api/sensors")] задає базовий маршрут для всіх методів цього контролера.

Усередині класу створюється приватне поле _db, яке містить контекст бази даних AppDbContext. Воно ініціалізується через конструктор за принципом dependency injection, тобто залежність передається автоматично через контейнер сервісів ASP.NET Core.

Основна логіка реалізована через метод GetHistory, який обробляє HTTP GET-запит за маршрутом api/sensors/history/{sensorId}. Цей метод приймає:

sensorId - ідентифікатор сенсора з URL;

limit - кількість записів (за замовчуванням 100), яка передається як query-параметр.

Метод робить запит до бази даних через Entity Framework Core. Спочатку з таблиці SensorData обираються всі записи, що відповідають потрібному sensorId. Далі вони сортуються за спаданням часу (OrderByDescending), щоб відібрати найновіші, після чого за допомогою Take(limit) залишається лише задана

кількість. Нарешті, результат упорядковується за зростанням часу (OrderBy), для того щоб відобразити дані в хронологічному порядку.

У фіналі метод повертає результат у форматі HTTP 200 OK разом зі списком даних сенсора.

Отже, цей контролер дає можливість отримати історію показників конкретного датчика, що використовується для аналізу, побудови графіків або перегляду подій у системі моніторингу.

AlertController.cs

Задача цього коду полягає в реалізації API-контролера AlertController, який відповідає за роботу з аварійними сповіщеннями у системі моніторингу: їх отримання та підтвердження (acknowledge).

Спочатку підключаються необхідні модулі: Data для доступу до бази даних, Services для бізнес-логіки роботи з тривогами, а також EntityFrameworkCore для виконання запитів до бази даних через ORM.

Далі оголошується контролер з атрибутами [ApiController] і [Route("api/alerts")]. Це означає, що всі запити до цього контролера будуть починатися з маршруту api/alerts, а ASP.NET Core автоматично оброблятиме HTTP-запити та валідацію.

Усередині класу оголошуються дві залежності: _db - контекст бази даних AppDbContext, який використовується для роботи з таблицею тривог, _alertService - сервіс бізнес-логіки, який відповідає за операції з підтвердженням тривог. Обидві залежності передаються конструктором за принципом dependency injection.

Метод GET GetAlerts обробляє запит GET api/alerts після чого повертає список аварійних сповіщень. Необов'язковий параметр acknowledged дає змогу фільтрувати результати: true - лише підтверджені тривоги, false - лише непідтверджені, null - усі без винятку. Усередині методу створюється запит до таблиці Alerts через AsQueryable(), що дозволяє гнучко додавати фільтри. Якщо параметр acknowledged було вказано, то в такому разі накладається фільтр за полем IsAcknowledged. Після чого дані сортуються за спаданням часу (Timestamp), щоб найновіші з’явилися першими, і завантажуються з бази через ToListAsync(). У кінці повертається HTTP-відповідь 200 OK зі списком тривог.

Метод POST Acknowledge обробляє запит POST api/alerts/{id}/acknowledge і використовується для підтвердження конкретного аварійного сповіщення за його ідентифікатором id. Усередині викликається метод _alertService.AcknowledgeAsync(id), який змінює стан алерта на (підтверджений). Якщо операція не вдалася, наприклад, алерт не знайдено, повертається помилка 404 Not Found. У разі успіху повертається 200 OK.

Data

AppDbContext.cs

Клас AppDbContext є ключовим компонентом для серверної частини програми системи аварійного моніторингу та виконує роль контексту бази даних, через який відбувається взаємодія зі сховищем інформації за допомогою Entity Framework Core.

На початку підключаються простори імен, в яких містяться спільні моделі даних (SensorData та Alert), а також бібліотека Microsoft.EntityFrameworkCore, яка забезпечує ORM-функціонал для роботи з базою даних без використання SQL-запитів. Сам клас AppDbContext наслідується від базового класу DbContext, що дає йому змогу керувати підключенням до бази даних, слідкувати за змінами об’єктів та

виконувати операції створення, читання, оновлення та видалення даних (CRUD).

Усередині класу визначено дві основні сутності бази даних у вигляді властивостей DbSet. Перша - SensorData, яка представлена таблицею з даними сенсорів і використовується для збереження та отримання показників температури, рівня диму та вібрацій. Друга - Alerts, відповідальна за зберігання аварійних подій та повідомлень системи моніторингу.

Конструктор AppDbContext приймає параметр DbContextOptions, який містить конфігурацію підключення до бази даних, та передає його до базового класу DbContext. Це дозволяє налаштовувати підключення (наприклад, до SQLite) через механізм dependency injection у ASP.NET Core.

У результаті AppDbContext виступає центральним елементом доступу до даних у системі, тим самим забезпечує зручну та структуровану роботу з інформацією про сенсори та аварійні сповіщення без необхідності використання прямого SQL-коду.

Hubs

MonitoringHub.cs

Клас MonitoringHub є центральним компонентом real-time взаємодії в системі аварійного моніторингу та реалізує серверний SignalR Hub, через який відбувається обмін даними між сенсорами та всіма підключеними клієнтами.

На початку підключаються необхідні залежності: AppDbContext для роботи з базою даних, AlertService для обробки та аналізу аварійних ситуацій, модель SensorData зі спільними моделями, а також бібліотека Microsoft.AspNetCore.SignalR, яка забезпечує функціональність Hub.

Клас MonitoringHub наслідується від Hub, що дозволяє йому працювати як точка обміну повідомленнями в режимі реального часу між сервером і клієнтами (WPF, веб або інші клієнти).

У конструктор класу через dependency injection передаються два основні компоненти: _db, який є контекстом бази даних і використовується для збереження даних сенсорів, та _alertService, який відповідає за аналіз отриманих даних і визначення аварійних ситуацій. Такий підхід дає чітке розділення відповідальності між компонентами системи та спрощує їх тестування і підтримку.

Метод SendSensorData викликається клієнтом (сенсором) для передачі нових показників на сервер. Після того як відбулось отримання даних сервер спочатку виводить повідомлення в консоль про те що він прийняв інформацію, після чого додає отримані дані до бази даних через SensorData.Add і зберігає зміни за допомогою SaveChangesAsync. Далі всі підключені клієнти отримують оновлення в реальному часі через взаємодію з SignalR (Clients.All.SendAsync("ReceiveSensorData", data)).

Після цього дані передаються до AlertService, який виконує їх аналіз в пошуках небезпечних значень. У випадках виявлення аварійних ситуацій створюється об’єкт alert, який теж розсилається всім клієнтам через подію ReceiveAlert, забезпечуючи миттєве інформування про загрозу.

Метод AcknowledgeAlert використовується для підтвердження аварійного сповіщення. У ньому викликається _alertService.AcknowledgeAsync(alertId), який змінює статус відповідного алерта на «підтверджений». Якщо операція виконана успішно, усім підключеним клієнтам надсилається повідомлення

AlertAcknowledged з ідентифікатором алерта, що дозволяє синхронізувати стан системи між усіма учасниками.

Тож MonitoringHub відіграє ключову роль у системі аварійного моніторингу, через те що забезпечує прийом даних від сенсорів у реальному часі, зберігає їх у базі даних, розповсюджує оновлення між клієнтами, робить аналіз на наявність аварійних ситуацій, а також керує статусом сповіщень. Отже, цей клас реалізує основний механізм реального часу та забезпечує його роботу та взаємодію на базі SignalR у системі моніторингу.

Services

AlertService

Сервіс AlertService несе відповідальність за аналіз даних з сенсорів, визначення рівня небезпеки та створення аварійних сповіщень у системі моніторингу. Для роботи сервісу використовуються контекст бази даних AppDbContext та спільні моделі SensorData, Alert і AlertSeverity. Через механізм dependency injection у клас передається екземпляр AppDbContext, що дозволяє сервісу взаємодіяти з таблицею сповіщень у базі даних.

Основним методом сервісу є EvaluateAsync, який аналізує отримані дані сенсора. Метод приймає об’єкт SensorData, після чого викликає функцію GetWorstSeverity, що визначає найвищий рівень небезпеки серед показників температури, диму та вібрації. Якщо всі параметри перебувають у межах норми (AlertSeverity.Normal), метод повертає null, і аварійне повідомлення не створюється.

У разі виявлення небезпечних значень, відбувається створення нового об’єкту Alert, у якому зберігаються ідентифікатор сенсора, назва зони підприємства, текст повідомлення, рівень небезпеки, статус підтвердження (IsAcknowledged

дорівнює false) та час створення аварійного повідомлення. Після цього сповіщення додається до бази даних через _db.Alerts.Add(alert) і зберігається методом SaveChangesAsync. Створений об’єкт повертається для подальшої передачі клієнтам через SignalR.

Метод AcknowledgeAsync застосовується для підтвердження аварійного повідомлення. Він знаходть алерт за його ідентифікатором через FindAsync. Якщо запис не знайдено, метод повертає false. Якщо алерт існує, його властивість IsAcknowledged змінюється на true, зміни зберігаються в базі даних, після чого метод повертає true.

Для того щоб визначити рівень небезпеки використовується метод GetWorstSeverity, який окремо здійснює аналіз температури, рівня диму та вібрації. Для кожного параметра викликаються спеціальні методи класифікації: ClassifyTemperature, ClassifySmoke та ClassifyVibration. Кожен з яких порівнює значення з установленими крайніми значеннями та повертає відповідний до них рівень небезпеки: Normal, Warning, Critical або Emergency. Наприклад, температура понад 50°C класифікується як Warning, понад 70°C - як Critical, а понад 90°C - як Emergency. Аналогічний принцип використовується для оцінювання рівня диму та вібрації.

Метод BuildMessage генерує текст аварійного повідомлення. Він перевіряє, які саме показники досягли рівня небезпеки, та додає їх до повідомлення. У результаті створюється текст у форматі: [Emergency] Серверна - Температура: 95.0°C, Дим: 82.0%.

Отож, AlertService реалізує основну бізнес-логіку системи аварійного моніторингу: аналізує показники сенсорів, визначає рівень загрози, створює аварійні повідомлення, зберігає їх у базі даних та забезпечує механізм підтвердження тривог.

3.4 Реалізація клієнта WPF

Converters

InvertBoolConvertor.cs

37

Даний код реалізує конвертер InvertBoolConverter для WPF-застосунку, який використовується для конвертації логічних значень типу bool під час прив’язки даних (Data Binding). На початку підключаються простори імен

System.Globalization та System.Windows.Data, необхідні для роботи механізму конвертації значень у WPF.

Клас InvertBoolConverter реалізує інтерфейс IValueConverter, що дає можливість автоматично перетворювати значення між джерелом даних і елементами інтерфейсу користувача під час виконання прив’язки (Binding). Основна задача цього конвертера в інверсії булевих значень.

Метод Convert виконується при передачі значення з ViewModel до інтерфейсу користувача. У ньому робиться перевірка, чи є отримане значення типу bool. Якщо це дійсно так, значення конвертується за допомогою логічного оператора !, тобто true перетворюється на false, а false - на true.

Метод ConvertBack працює аналогічним чином, але використовується для перетворення в інший бік - під час передачі значення з інтерфейсу користувача назад до ViewModel. Він також виконує інверсію логічного значення.

Подібний конвертер має широкий обсяг застосування у WPF-застосунках для керування станами елементів інтерфейсу. Наприклад, він може використовуватись для інверсії властивості IsEnabled, блокування або

приховування елементів, зміни стану кнопок, а також для роботи з властивостями Visibility або IsChecked.

Приклад практичного ивикористання можна описати так: кнопка може автоматично ставати неактивною, коли змінна IsConnected має значення true. У такому випадку конвертер виконує інверсію значення без необхідності створення додаткової логіки у ViewModel.

Таким чином, InvertBoolConverter забезпечує простий і зручний механізм конвертації булевих значень у WPF-прив’язках даних, що спрощує реалізацію логіки взаємодії між інтерфейсом користувача та моделлю представлення.

Models

SensorCardModel.cs

Цей код реалізує модель SensorCardModel для WPF-застосунку системи аварійного моніторингу, яка відповідає за зберігання, обробку та відображення даних про сенсор у користувацькому інтерфейсі, а також за автоматичне оновлення стану при зміні показників.

На початку файлу підключаються необхідні простори імен: EmergencyMonitoring.Shared.Models, що містить спільні моделі та перелік AlertSeverity; System.ComponentModel, який забезпечує механізм повідомлення про зміну властивостей; System.Runtime.CompilerServices, що використовується для автоматичного визначення імені властивості; та System.Windows.Media, який надає можливість роботи з кольорами у WPF.

Клас SensorCardModel реалізовує інтерфейс INotifyPropertyChanged, який є ключовим елементом Data Binding у WPF. Завдяки цьому інтерфейсу, інтерфейс користувача автоматично оновлюється кожного разу, як змінюються значення властивостей моделі.

У класі оголошується подія PropertyChanged, яка починає працювати при зміні будь-якої властивості. Далі визначаються приватні поля та відповідні публічні властивості: SensorId (ідентифікатор сенсора), Zone (назва контрольованої зони), Temperature (температура), SmokeLevel (рівень диму) та Vibration (рівень вібрації). У сетерах цих властивостей відбувається виклик методу OnPropertyChanged(), який повідомляє WPF про зміну значення та оновлює інтерфейс. Для параметрів температури, диму та вібрації також додатково викликається метод RefreshStatus(), оскільки їх зміна впливає на загальний рівень небезпеки.

Окремо визначені властивості Status та StatusColor, відповідають за текстовий опис стану сенсора та його візуальне відображення відповідним кольором. За замовчуванням встановлюється стан (Норма) та зелений колір (LimeGreen).

Метод RefreshStatus() відповідає за оновлення статусу сенсора. Він викликає метод GetWorstSeverity(), який визначає найвищий рівень небезпеки серед усіх показників. Далі через конструкцію switch встановлюються відповідні значення статусу та кольору: для Emergency - червоний колір і повідомлення “Надзвичайна ситуація”, для Critical - помаранчево-червоний, для Warning - помаранчевий, а для Normal - зелений. Після цього викликається OnPropertyChanged для властивостей Status і StatusColor, що забезпечує автоматичне оновлення інтерфейсу користувача.

Метод GetWorstSeverity() реалізовує логіку оцінки рівня небезпеки аналізуючи три параметра: температуру, рівень диму та вібрацію. Кожен з яких порівнюється з крайніми значеннями та отримує відповідний до цього значення рівень (Normal, Warning, Critical або Emergency). Після цього визначається найвищий рівень небезпеки за допомогою Math.Max.

Метод OnPropertyChanged, який викликає подію PropertyChanged, відповідає за сповіщення WPF про зміну властивостей. Завдяки атрибуту

[CallerMemberName] ім’я властивості передається автоматично, і не потрібно вказувати його вручну.

Маємо: SensorCardModel ключова модель представлення сенсора у WPF-застосунку, яка забезпечує автоматичне оновлення інтерфейсу, аналіз стану сенсора та відображення рівня небезпеки в режимі реального часу.

Models

AlertModel.cs

Код реалізує модель AlertModel у WPF-застосунку у системі аварійного моніторингу, яка відповідає за відображення аварійних сповіщень у інтерфейсі користувача та забезпечує автоматичне оновлення даних у разі їх зміни. На

початку підключаються необхідні для роботи простори імен: EmergencyMonitoring.Shared.Models, що містить спільні моделі та перелік AlertSeverity, а також System.ComponentModel і System.Runtime.CompilerServices, які використовуються для реалізації INotifyPropertyChanged та автоматичного визначення імен властивостей.

Клас AlertModel реалізує інтерфейс INotifyPropertyChanged, що є основою Data Binding у WPF і забезпечує автоматичне оновлення інтерфейсу користувача при зміні властивостей моделі. У класі оголошується подія PropertyChanged, яка спрацьовує кожного разу при зміні будь-якої властивості.

Далі задаються незмінні (init-only) властивості, які встановлюються лише під час створення об’єкта і більше не можуть змінюватися. До них належать: Id (ідентифікатор алерта), SensorId (ідентифікатор сенсора), Zone (зона виникнення), Message (текст повідомлення), Severity (рівень небезпеки) та Timestamp (час створення). Використання ініціалізатора init гарантує що ці значення не будуть змінюватись після ініціалізації об’єкта.

Окремо реалізовано властивість IsAcknowledged, яка відображає стан підтвердження аварійного повідомлення. Вона працює на основі приватного поля _isAcknowledged, і при зміні значення викликається метод OnPropertyChanged, що забезпечує оновлення даних у інтерфейсі користувача. Крім того, додатково оновлюється властивість AcknowledgeLabel, оскільки її значення напряму залежить від стану підтвердження.

Властивість AcknowledgeLabel обчислювальна і повертає текст, який відображається в інтерфейсі користувача: якщо алерт уже підтверджено - “Підтверджено”, якщо ні - “Підтвердити”. Таким чином забезпечується динамічна зміна тексту кнопки в UI відповідно до поточного стану об’єкта.

Властивість SeverityDisplay також є обчислюваною і перетворює технічний рівень небезпеки (AlertSeverity) у зрозумілий текст для користувача. Наприклад, Warning відображається як “Попередження”, Critical - як “Критично”, а Emergency - як “Надзвичайна ситуація”.

Метод FromAlert грає роль перетворення об’єкта доменної моделі Alert у AlertModel, який застосовується у WPF. Він копіює всі потрібні поля з серверної моделі в модель представлення, гарантуючи зручну інтеграцію між backend та інтерфейсом користувача.

В свою чергу метод OnPropertyChanged відповідає за сповіщення WPF про зміну властивостей. Завдяки атрибуту [CallerMemberName] ім’я властивості визначається автоматично, що спрощує його виклик.

Таким чином, AlertModel є моделлю представлення аварійних сповіщень у WPF-застосунку, яка забезпечує відображення інформації про тривоги, їхній статус підтвердження, рівень небезпеки та автоматичне оновлення інтерфейсу в режимі реального часу.

Services

ApiService.cs

Файл ApiService у WPF-застосунку системи аварійного моніторингу, відповідає за взаємодію з серверним REST API. Його основною задачею є отримання історії даних сенсорів, завантажених до списку аварійних сповіщень та відправка запитів для підтвердження тривог.

На початку підключаються простори імен EmergencyMonitoring.Shared.Models, які містять спільні моделі даних (SensorData, Alert), а також System.Net.Http і System.Net.Http.Json, що використовуються для виконання HTTP-запитів і автоматичної десеріалізації JSON-відповідей.

У класі ApiService створюється приватне поле _http типу HttpClient, яке відповідає за всі HTTP-запити до сервера. У налаштуваннях цього поля задається

BaseAddress, базова адреса API-сервера http://localhost:5223, що дозволяє не дублювати повні URL у кожному запиті.

В свою чергу метод GetHistoryAsync використовується для отримання історії даних конкретного сенсора. Він приймає sensorId та необов’язковий параметр limit, який визначає кількість записів. Усередині виконується HTTP GET-запит до ендпоінта /api/sensors/history/{sensorId} з параметром limit. Отримана JSON-відповідь автоматично перетворюється у список об’єктів SensorData за допомогою GetFromJsonAsync, якщо результат буде null повертається порожній список.

GetAlertsAsync відповідає за отримання списку аварійних сповіщень. Він підтримує необов’язковий фільтр acknowledged, який дозволяє отримувати або всі тривоги, або лише підтверджені чи непідтверджені. Якщо параметр заданий, формується URL із query-параметром acknowledged, інакше використовується

базовий ендпоінт /api/alerts. Результат також десеріалізується у список об’єктів Alert, а якщо сервер повертає значення null, в такому разі метод повертає порожній список.

Метод AcknowledgeAlertAsync застосовується для того щоб підтвердити аварійне сповіщення. Його задача відправити HTTP POST-запит на ендпоінт /api/alerts/{id}/acknowledge без тіла запиту. У результаті сервер оновлює статус відповідного алерта на “підтверджений”.

Отже маємо, ApiService виконує роль посередника між WPF-клієнтом і сервером, забезпечуючи зручну роботу з REST API, отримання актуальних даних у реальному часі та керування станом аварійних сповіщень.

Services

SignalRService.cs

Суть цього коду у реалізазії сервісу SignalRService у WPF-застосунку системи аварійного моніторингу, Цей сервіс відповідає за реальний обмін даними з сервером у режимі реального часу за допомогою SignalR.

підключаються простори імен EmergencyMonitoring.Shared.Models, які містять спільні моделі даних (SensorData, Alert), та Microsoft.AspNetCore.SignalR.Client, що надає можливість створювати клієнтське підключення до SignalR-хабу.

У класі SignalRService оголошується приватне поле _connection типу HubConnection?, для зберігання активного підключення до серверного SignalR-хабу. Воно може мати значення null, якщо з’єднання ще не ініціалізовано.

Також визначені події:

OnSensorData - викликається при отриманні нових даних сенсора;

OnAlert - викликається при прийнятті аварійного сповіщення;

OnAlertAcknowledged - викликається при підтвердженні тривоги.

Ці події дозволяють іншим частинам WPF-додатку реагувати на оновлення даних у реальному часі.

Метод StartAsync відповідальний за ініціалізацію та запуск SignalR-з'єднання. Першим створюється об'єкт HubConnection за допомогою HubConnectionBuilder, де задається адреса сервера http://localhost:5223/monitoringHub та вмикається автоматичне перепідключення через WithAutomaticReconnect().

Далі налаштовуються обробники вхідних повідомлень від сервера:

ReceiveSensorData - отримує об'єкт SensorData і викликає подію OnSensorData;

ReceiveAlert – отримує доступ до об'єкта Alert і викликає подію OnAlert;

AlertAcknowledged - отримує ID тривоги і викликає подію OnAlertAcknowledged.

Налаштувавши обробники викликається StartAsync(), який встановлює активне підключення до SignalR-хабу.

Метод AcknowledgeAlertAsync відправляє команди на сервер про підтвердження аварійного сповіщення. Перед викликом перевіряється, чи з'єднання активне (HubConnectionState.Connected). Якщо так, викликається серверний метод AcknowledgeAlert через InvokeAsync, і на сервер передається alertId.

Таким чином, SignalRService забезпечує двосторонню взаємодію у реальному часі між WPF-клієнтом і сервером: отримує оновлення про сенсори та тривоги, а також дозволяє користувачу підтверджувати аварійні сповіщення в режимі реального часу.

ViewModels

AlertsViewModel.cs

Реалізує AlertsViewModel у WPF-застосунку цієї системи аварійного моніторингу і відповідає за керування списком тривог, їх завантаження, оновлення в реальному часі та взаємодію з користувачем.

Підключення потрібних просторів імен: EmergencyMonitoring.Shared.Models, що містить спільні моделі даних, EmergencyMonitoring.WPF.Models для UI-моделей (AlertModel), а також сервіси ApiService і SignalRService. Додатково використовуються ObservableCollection для автоматичного оновлення інтерфейсу, ICommand для реалізації команд у MVVM та Dispatcher для безпечного оновлення UI з інших потоків.

Клас AlertsViewModel містить колекцію Alerts, що зберігає список тривог і автоматично оновлює інтерфейс при зміні даних завдяки використанню ObservableCollection.

Також оголошено команду AcknowledgeCommand, яка використовується для підтвердження аварійного сповіщення користувачем.

До конструктору AlertsViewModel через dependency injection передаються ApiService, SignalRService та Dispatcher. Вони зберігаються у приватних полях і використовуються для роботи з API, отримання повідомлень та безпечного оновлення інтерфейсу у реальному часі.

Усередині конструктора ініціалізується AcknowledgeCommand за допомогою RelayCommand. Команда приймає об’єкт AlertModel і виконує асинхронну операцію: якщо модель не null і тривога ще не була підтверджена (IsAcknowledged дорівнює false), викликається метод _signalR.AcknowledgeAlertAsync(m.Id), який надсилає запит на сервер для підтвердження тривоги.

Стосовно методу LoadAsync він відповідає за початкове завантаження списку тривог із сервера. Також викликає _api.GetAlertsAsync(), після чого через Dispatcher.Invoke очищає локальну колекцію Alerts і додає нові елементи, перетворюючи їх у AlertModel за допомогою FromAlert. Використання Dispatcher гарантує безпечне оновлення UI-потоку.

Метод HandleNewAlert потрібен для обробки нових аварійних сповіщень, які з’являються у реальному часі через SignalR. Нова тривога додається на початок списку (Insert(0, ...)), щоб він одразу відображався першим у інтерфейсі.

Метод HandleAcknowledged обробляє підтвердження тривоги. Знаходить відповідний елемент у колекції за alertId і, якщо знаходить його, встановлює IsAcknowledged дорвнює true, що автоматично оновлює інтерфейс завдяки меха Data Binding.

AlertsViewModel є важливим компонентом MVVM-архітектури, який забезпечує завантаження історичних тривог із API, отримання нових сповіщень у реальному часі через SignalR, підтвердження тривог користувачем та автоматичне оновлення інтерфейсу за допомогою ObservableCollection і Dispatcher.

ViewModels

DashbordViewModel.cs

Цей код потрібен для реалізації DashboardViewModel у WPF-застосунку розробленої системи аварійного моніторингу, адже він відповідає за відображення та оновлення списку сенсорів у реальному часі на головній панелі (dashboard).

Підключені необхідні простори імен: EmergencyMonitoring.Shared.Models для роботи з даними сенсорів, EmergencyMonitoring.WPF.Models для UI-моделі SensorCardModel, а також ObservableCollection і Dispatcher, які гарантують автоматичне оновлення інтерфейсу та безпечну роботу з UI-потоком.

У класі DashboardViewModel оголошується властивість Sensors - це колекція ObservableCollection<SensorCardModel>, яка містить список сенсорів і автоматично оновлює інтерфейс WPF при додаванні або зміні елементів.

Також визначається приватне поле _dispatcher, що потрібне для виконання змін у UI-потоці, оскільки у WPF оновлення інтерфейсу з фонового потоку є забороненим.

В конструкторі DashboardViewModel через dependency injection передається Dispatcher, і зберігається у приватному полі для його використання в подальшому.

Основний метод класу - HandleSensorData(SensorData data), викликається при надходженні нових даних від сенсорів, наприклад через SignalR.

В методі використовується _dispatcher.Invoke, щоб гарантувати виконання коду в UI-потоці. Потім виконується пошук сенсора в колекції Sensors за SensorId.

Якщо сенсор вже існує - його значення оновлюються. Якщо ще не існує - створюється новий об’єкт SensorCardModel, заповнюються поля SensorId і Zone, після чого він додається до колекції.

Після цього оновлюються параметри сенсора: Temperature, SmokeLevel і Vibration. Що автоматично запускає оновлення інтерфейсу завдяки механізму INotifyPropertyChanged у SensorCardModel.

DashboardViewModel забезпечує динамічне оновлення списку сенсорів, додавання нових сенсорів у реальному часі, оновлення показників існуючих сенсорів і безпечну роботу з UI через Dispatcher.

VewModels

HistoryViewModel.cs

HistoryViewModel у WPF-застосунку в даній системі аварійного моніторингу відповідає за відображення історичних даних сенсорів у вигляді графіків. Його основне призначення - отримання даних з API та побудова графіків температури, рівня диму та вібрації у часі за допомогою бібліотеки OxyPlot.

Спочатку підключаються необхідні простори імен: ApiService для отримання даних із сервера, компоненти OxyPlot (Axes, Series) для побудови графіків, а також ObservableCollection і Dispatcher для роботи з UI. Інтерфейс INotifyPropertyChanged використовується для автоматичного оновлення інтерфейсу при зміні властивостей.

У класі оголошується подія PropertyChanged, яка є частиною MVVM-механізму і забезпечує реакцію WPF на зміни даних. Далі визначаються основні залежності: _api для роботи з серверним API та _dispatcher для безпечного оновлення UI з фонового потоку.

Властивість SensorIds зберігає список доступних сенсорів (SENSOR-1 до SENSOR-5), які користувач може обрати для перегляду історії. SelectedSensorId зберігає поточний вибраний сенсор і при зміні викликає OnPropertyChanged, що оновлює прив’язку у UI. Властивість PlotModel представляє сам графік і містить усі його налаштування та дані.

Команда RefreshCommand потрібна тут для оновлення графіка і викликає завантаження історичних даних. У конструкторі через dependency injection передаються ApiService і Dispatcher, після чого ініціалізується команда та викликається InitPlotModel, який створює базову конфігурацію графіка.

Метод InitPlotModel налаштовує вигляд графіка: встановлює заголовок, темний фон, білий текст і стилі. Також додаються дві осі - горизонтальна вісь часу

(DateTimeAxis) і вертикальна числова вісь (LinearAxis). Окрім цього налаштовується легенда, яка пояснює значення кожної лінії.

Основний метод LoadHistoryAsync несе відповідальність за завантаження історичних даних вибраного сенсора. Він отримує останні 50 записів через API, після чого через Dispatcher.Invoke оновлює UI-потік. Старі дані графіка очищаються, і створюються три серії: температура, дим і вібрація.

Далі кожен запис обробляється у циклі: час перетворюється у формат для графіка, а відповідні значення додаються до серій. Після цього серії додаються до графіка і викликається InvalidatePlot(true), що оновлює відображення візуалізації.

Метод OnPropertyChanged завершує клас і забезпечує повідомлення WPF про зміну властивостей, що гарантує автоматичне оновлення інтерфейсу.

Таким чином, HistoryViewModel виконує роль аналітичного модуля системи, який завантажує історичні дані сенсорів та візуалізує їх у вигляді інтерактивних графіків у реальному часі.

ViewModels

MainViewModel.cs

Даний код MainViewModel у WPF-застосунку системи аварійного моніторингу є базовим ViewModel, який відповідає за керування над відображеним екраном (View) у застосунку.

На початку підключаються простори імен System.ComponentModel та System.Runtime.CompilerServices, які забезпечують реалізацію механізму INotifyPropertyChanged для автоматичного оновлення інтерфейсу.

Клас MainViewModel реалізує інтерфейс INotifyPropertyChanged, що є ключовим елементом MVVM-архітектури. Завдяки цьому WPF автоматично реагує на зміну властивостей у ViewModel і оновлює інтерфейс користувача.

У класі оголошується подія PropertyChanged, яка спрацьовує щоразу при зміні значення будь-якої з властивостей.

Основною властивістю класу є CurrentView, вона зберігає поточну активну View (екран або компонент інтерфейсу). А також має приватне поле _currentView, а при зміні значення викликається OnPropertyChanged(), що повідомляє UI про оновлення.

Ця властивість використовується для реалізації навігації в застосунку - наприклад, перемикання між Dashboard, Alerts або History екранами без перезавантаження програми.

Метод OnPropertyChanged відповідає за сповіщення інтерфейсу про зміну властивостей. Атрибут [CallerMemberName] автоматично підставляє ім’я властивості, яка була змінена, що спрощує код і зменшує шанс на помилку.

Тобто, MainViewModel виконує роль центрального контролера навігації у WPF-застосунку, гарантуючи зміну активного екрану та автоматичне оновлення інтерфейсу в рамках MVVM-архітектури.

ViewModels

RelayCommand.cs

Даний код реалізує два класи RelayCommand і RelayCommand<T> у WPF-застосунку для системи аварійного моніторингу. Вони використовуються для

реалізації команд у MVVM-архітектурі та забезпечують зв’язок між інтерфейсом користувача та логікою ViewModel без використання code-behind.

Обидва класи реалізують інтерфейс ICommand, який у WPF відповідає за виконання дій з інтерфейсу, перевірку доступності команди через CanExecute та її запуск через Execute.

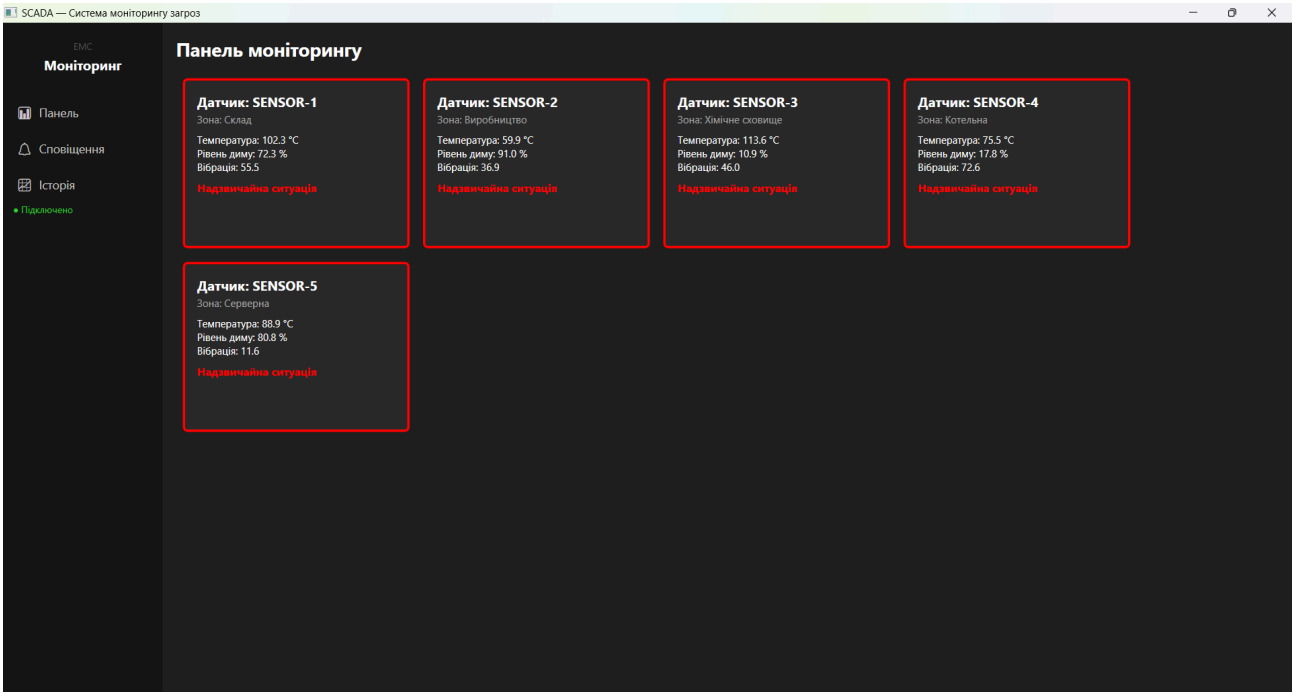
RelayCommand (без параметрів) містить асинхронну функцію _execute, яка виконується при виклику команди, та прапорець _isExecuting, що запобігає повторному запуску команди під час її виконання. У методі CanExecute перевіряється стан _isExecuting: якщо команда вже виконується, вона стає недоступною. Метод Execute спочатку встановлює _isExecuting дорівнює true, викликає CanExecuteChanged для оновлення UI, виконує асинхронну операцію _execute(), а після завершення повертає _isExecuting у false і знову оновлює стан команди. Що дає змогу блокувати кнопку або інший елемент інтерфейсу під час виконання операції.

RelayCommand<T> є узагальненою версією команди, яка приймає параметр типу T. Вона зберігає функцію _execute, яка виконується з переданим параметром. Метод CanExecute завжди повертає true, тобто команда завжди доступна. У методі Execute перевіряється тип переданого параметра: якщо він відповідає T, він передається у активну функцію, інакше використовується значення за замовчуванням.

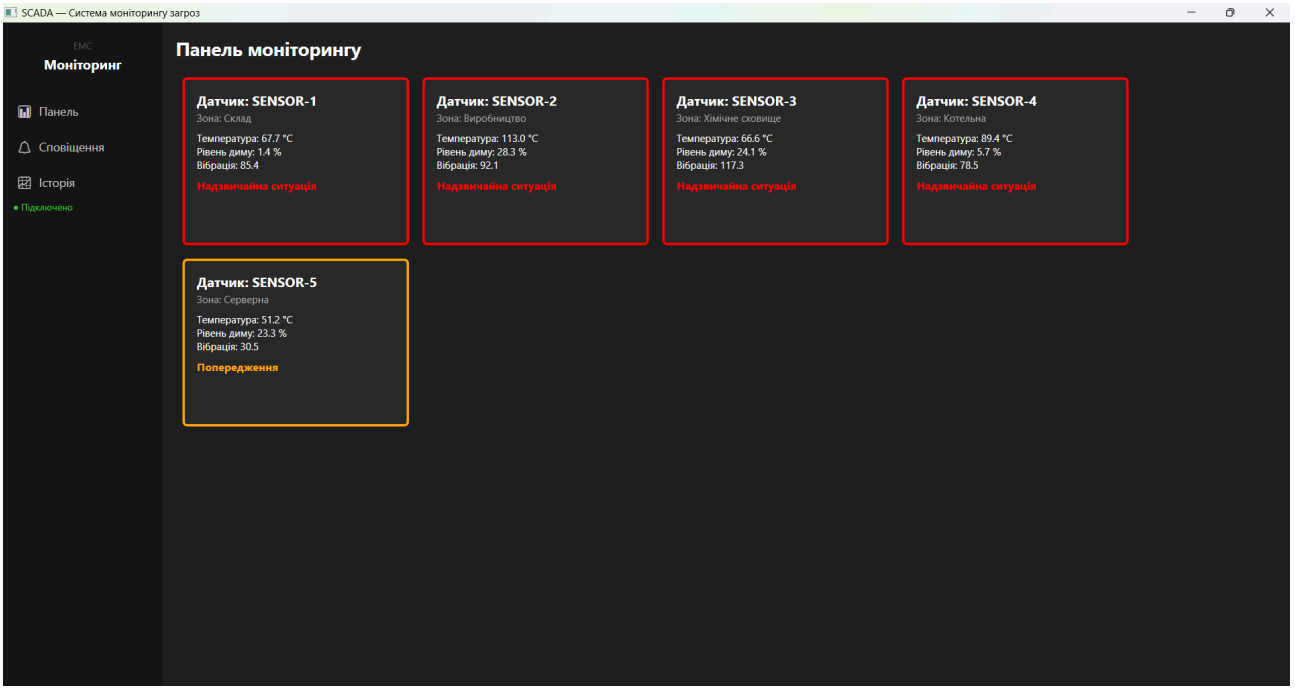
Отже, RelayCommand і RelayCommand<T> забезпечують механізм реалізації команд у WPF. Вони дозволяють виконувати асинхронні дії, передавати параметри у ViewModel та контролювати стан виконання операцій, що є ключовим елементом MVVM-архітектури.

3.5 Інтерфейс користувача

Отже відкривши програму користувач відразу побачить темний фон і як на ньому генеруються дані а також рівень небезпеки в кожній із зон підприємства на даний момент. Кожна зона оснащена датчиком п’ять зон і стільки ж датчиків по одному на зону, Кожен датчик буде відображати в межах зони свого прямокутника значення які притаманні для його зони, а також рівень тривоги, на кадрі [3.1] нижче в кожній зоні відображається напис “надзвичайна ситуація” що говорить нам у даний момент в кожній зоні аварія. А на кадрі [3.2] у п’ятій зоні тільки попередження що може виникнути аварія, у той час як на всіх інших зонах уже виникла аварія.



Кадр. 3.1 – відображення датчиків аварії



Кадр. 3.2 – на п’ятому датчику попередження

Висновки

У третьому розділі головний акцент та і сам увесь розділ був присвячений спробі описати код усієї цієї програми, щоб зрозуміти як вона працює під капотом. Було пару провальних спроб у створенні але з рештою вийшла працююча програма. Ця програма задумана як прототип що означає що поставлену задачу “прототип сенсорної мережі для моніторингу надзвичайних ситуацій на підприємстві” вдалось реалізувати, адже програма здатна відображати аварії також здатна показувати графіки і історію аварій.

Використані Джерела

1) https://www.vpnunlimited.com/ua/help/cybersecurity/security-monitoring-systems?srsltid=AfmBOoqiTDelF9jCcBADwuj2-5QzFTNQIgvS-HFFqchelF7MG966i5C4

2) https://www.oracle.com/internet-of-things/

3) https://weagro.ua/blog/internet-rechej-iot-shho-cze-ta-jogo-vykorystannya-v-silskomu-gospodarstvi

4)https://training.qatestlab.com/blog/technical-articles/client-server-architecture

5) https://www.geeksforgeeks.org/system-design/client-server-model/

6)https://dotnetfullstackdev.medium.com/real-time-data-transfer-with-websockets-and-signalr-in-net-core-409b0d50719b

7) https://appmaster.io/ru/blog/arkhitektura-sistem-real-nogo-vremeni-websockets-signalr

8) https://dev.to/chakewitz/building-real-time-applications-with-signalr-and-c-m5a

9) https://wezom.com.ua/ua/blog/bazi-danih-viznachennya-ta-riznovidi

10) https://university.sigma.software/what-is-database/

11) https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes

12) https://arxiv.org/abs/1803.04780?utm_source

13) https://introserv.com/ua/blog/what-is-a-server-and-how-does-it-differ-from-a-computer/

14) https://www.maxzosim.com/data-modelling/

15) https://learn.microsoft.com/ru-ru/dotnet/desktop/wpf/app-development/deploying-a-wpf-application-wpf

16) https://dou.ua/lenta/articles/sanky-diagram/

Додаток А

Server

Program.cs

```
using EmergencyMonitoring.Server.Data;
using EmergencyMonitoring.Server.Hubs;
using EmergencyMonitoring.Server.Services;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSignalR();

builder.Services.AddDbContext<AppDbContext>(options =>
{
    options.UseSqlite("Data Source=emergency.db");
});

builder.Services.AddScoped<AlertService>();

builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(policy =>
    {
        policy
            .AllowAnyHeader()
            .AllowAnyMethod()
            .AllowCredentials()
            .SetIsOriginAllowed(_ => true);
    });
});

var app = builder.Build();

app.UseCors();
app.MapControllers();
app.MapHub<MonitoringHub>("/monitoringHub");

app.Run();
```

SensorController.cs

```
using EmergencyMonitoring.Server.Data;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace EmergencyMonitoring.Server.Controllers;

[ApiController]
[Route("api/sensors")]
public class SensorController : ControllerBase
{
    private readonly AppDbContext _db;

    public SensorController(AppDbContext db)
    {
        _db = db;
    }
```

```
    [HttpGet("history/{sensorId}")]
    public async Task<IActionResult> GetHistory(string sensorId, [FromQuery] int limit =
100)
    {
        var data = await _db.SensorData
            .Where(x => x.SensorId == sensorId)
            .OrderByDescending(x => x.Timestamp)
            .Take(limit)
            .OrderBy(x => x.Timestamp)
            .ToListAsync();

        return Ok(data);
    }
}
```

AlertController.cs

```
using EmergencyMonitoring.Server.Data;
using EmergencyMonitoring.Server.Services;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace EmergencyMonitoring.Server.Controllers;

[ApiController]
[Route("api/alerts")]
public class AlertController : ControllerBase
{
    private readonly AppDbContext _db;
    private readonly AlertService _alertService;

    public AlertController(AppDbContext db, AlertService alertService)
    {
        _db = db;
        _alertService = alertService;
    }

    [HttpGet]
    public async Task<IActionResult> GetAlerts([FromQuery] bool? acknowledged = null)
    {
        var query = _db.Alerts.AsQueryable();

        if (acknowledged.HasValue)
            query = query.Where(a => a.IsAcknowledged == acknowledged.Value);

        var alerts = await query
            .OrderByDescending(a => a.Timestamp)
            .ToListAsync();

        return Ok(alerts);
    }

    [HttpPost("{id}/acknowledge")]
    public async Task<IActionResult> Acknowledge(int id)
    {
        var success = await _alertService.AcknowledgeAsync(id);
        if (!success) return NotFound();
        return Ok();
    }
}
```

Data

AppDbContext.cs

```
using EmergencyMonitoring.Shared.Models;
using Microsoft.EntityFrameworkCore;

namespace EmergencyMonitoring.Server.Data;

public class AppDbContext : DbContext
{
    public DbSet<SensorData> SensorData => Set<SensorData>();
    public DbSet<Alert> Alerts => Set<Alert>();

    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }
}
```

Hubs

MonitoringHub.cs

```
using EmergencyMonitoring.Server.Data;
using EmergencyMonitoring.Server.Services;
using EmergencyMonitoring.Shared.Models;
using Microsoft.AspNetCore.SignalR;

namespace EmergencyMonitoring.Server.Hubs;

public class MonitoringHub : Hub
{
    private readonly AppDbContext _db;
    private readonly AlertService _alertService;

    public MonitoringHub(AppDbContext db, AlertService alertService)
    {
        _db = db;
        _alertService = alertService;
    }

    public async Task SendSensorData(SensorData data)
    {
        Console.WriteLine($"SERVER RECEIVED: {data.SensorId}");

        _db.SensorData.Add(data);
        await _db.SaveChangesAsync();

        await Clients.All.SendAsync("ReceiveSensorData", data);

        var alert = await _alertService.EvaluateAsync(data);
        if (alert != null)
            await Clients.All.SendAsync("ReceiveAlert", alert);
    }

    public async Task AcknowledgeAlert(int alertId)
    {
        var success = await _alertService.AcknowledgeAsync(alertId);
        if (success)
            await Clients.All.SendAsync("AlertAcknowledged", alertId);
    }
}
```

Services

AlertService.cs

```
using EmergencyMonitoring.Server.Data;
using EmergencyMonitoring.Shared.Models;

namespace EmergencyMonitoring.Server.Services;

public class AlertService
{
    private readonly AppDbContext _db;

    public AlertService(AppDbContext db)
    {
        _db = db;
    }

    public async Task<Alert?> EvaluateAsync(SensorData data)
    {
        var severity = GetWorstSeverity(data);
        if (severity == AlertSeverity.Normal)
            return null;

        var alert = new Alert
        {
            SensorId = data.SensorId,
            Zone = data.Zone,
            Message = BuildMessage(data, severity),
            Severity = severity,
            IsAcknowledged = false,
            Timestamp = DateTime.UtcNow
        };

        _db.Alerts.Add(alert);
        await _db.SaveChangesAsync();
        return alert;
    }

    public async Task<bool> AcknowledgeAsync(int alertId)
    {
        var alert = await _db.Alerts.FindAsync(alertId);
        if (alert == null) return false;

        alert.IsAcknowledged = true;
        await _db.SaveChangesAsync();
        return true;
    }

    private static AlertSeverity GetWorstSeverity(SensorData data)
    {
        var t = ClassifyTemperature(data.Temperature);
        var s = ClassifySmoke(data.SmokeLevel);
        var v = ClassifyVibration(data.Vibration);
        return (AlertSeverity)Math.Max(Math.Max((int)t, (int)s), (int)v);
    }

    private static AlertSeverity ClassifyTemperature(double v) =>
        v > 90 ? AlertSeverity.Emergency :
        v > 70 ? AlertSeverity.Critical :
        v > 50 ? AlertSeverity.Warning : AlertSeverity.Normal;

    private static AlertSeverity ClassifySmoke(double v) =>
        v > 80 ? AlertSeverity.Emergency :
        v > 50 ? AlertSeverity.Critical :
        v > 20 ? AlertSeverity.Warning : AlertSeverity.Normal;
```

```
    private static AlertSeverity ClassifyVibration(double v) =>
        v > 60 ? AlertSeverity.Emergency :
        v > 40 ? AlertSeverity.Critical :
        v > 20 ? AlertSeverity.Warning : AlertSeverity.Normal;

    private static string BuildMessage(SensorData data, AlertSeverity severity)
    {
        var parts = new List<string>();
        if (ClassifyTemperature(data.Temperature) >= severity)
            parts.Add($"Температура: {data.Temperature:F1}°C");
        if (ClassifySmoke(data.SmokeLevel) >= severity)
            parts.Add($"Дим: {data.SmokeLevel:F1}%");
        if (ClassifyVibration(data.Vibration) >= severity)
            parts.Add($"Вібрація: {data.Vibration:F1}");
        return $"[{severity}] {data.Zone} — {string.Join(", ", parts)}";
    }
}
```

SensorSimulator

Program.cs

```
using EmergencyMonitoring.Shared.Models;
using Microsoft.AspNetCore.SignalR.Client;

var connection = new HubConnectionBuilder()
    .WithUrl("http://localhost:5223/monitoringHub")
    .WithAutomaticReconnect()
    .Build();

await connection.StartAsync();

var sensors = new[]
{
    ("SENSOR-1", "Склад",            ConsoleColor.Cyan),
    ("SENSOR-2", "Виробництво",      ConsoleColor.Yellow),
    ("SENSOR-3", "Хімічне сховище",  ConsoleColor.Green),
    ("SENSOR-4", "Котельна",         ConsoleColor.Magenta),
    ("SENSOR-5", "Серверна",         ConsoleColor.Blue),
};

var consoleLock = new object();

static (double temp, double smoke, double vib) GenerateValues(Random rng) => (
    Math.Round(rng.NextDouble() * 115 + 15, 1),   // Temperature  15 – 130 °C
    Math.Round(rng.NextDouble() * 100,       1),   // Smoke level   0 – 100 %
    Math.Round(rng.NextDouble() * 120,       1)    // Vibration     0 – 120
);

var cts   = new CancellationTokenSource();
var token = cts.Token;

var sensorTasks = sensors.Select(async s =>
{
    var (sensorId, zone, color) = s;
    var rng = new Random();

    while (!token.IsCancellationRequested)
    {
        try
        {
            var (temp, smoke, vib) = GenerateValues(rng);

            var data = new SensorData
            {
                SensorId    = sensorId,
                Zone        = zone,
                Temperature = temp,
```

```
                SmokeLevel  = smoke,
                Vibration   = vib,
                Timestamp   = DateTime.Now
            };

            await connection.SendAsync("SendSensorData", data, token);

            lock (consoleLock)
            {
                Console.ForegroundColor = color;
                Console.WriteLine(
                    $"[{DateTime.Now:HH:mm:ss}]  {sensorId,-10} ({zone,-18})" +
                    $"  Темп: {temp,6:F1} °C" +
                    $"  Дим: {smoke,6:F1} %" +
                    $"  Вібрація: {vib,6:F1}");
                Console.ResetColor();
            }
        }
        catch (OperationCanceledException) { break; }
        catch (Exception ex)
        {
            lock (consoleLock)
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine($"Помилка {sensorId}: {ex.Message}");
                Console.ResetColor();
            }
        }

        await Task.Delay(5000, token).ContinueWith(_ => { });
    }
}).ToArray();

Console.Clear();
Console.WriteLine("╔══════════════════════════════════════════════════════════╗");
Console.WriteLine("║        СИМУЛЯТОР ДАТЧИКІВ АВАРІЙНОГО МОНІТОРИНГУ         ║");
Console.WriteLine("╚══════════════════════════════════════════════════════════╝");
Console.WriteLine();
Console.WriteLine($"  Датчиків активно: {sensors.Length}   |   Інтервал: 5 сек   |
Натисніть [0] для виходу");
Console.WriteLine();

foreach (var (sensorId, zone, color) in sensors)
{
    Console.ForegroundColor = color;
    Console.WriteLine($"  ■  {sensorId}  –  {zone}");
    Console.ResetColor();
}

Console.WriteLine();
Console.WriteLine("──────────────────────────────────────────────────────────");
Console.WriteLine();

while (true)
{
    var key = Console.ReadKey(intercept: true).KeyChar;
    if (key == '0')
    {
        await cts.CancelAsync();
        await Task.WhenAll(sensorTasks);
        return;
    }
}
```

Shared

Molels

Alert.cs

```
namespace EmergencyMonitoring.Shared.Models;

public class Alert
{
    public int Id { get; set; }
    public string SensorId { get; set; } = string.Empty;
    public string Zone { get; set; } = string.Empty;
    public string Message { get; set; } = string.Empty;
    public AlertSeverity Severity { get; set; }
    public bool IsAcknowledged { get; set; }
    public DateTime Timestamp { get; set; }
}
```

AlertSeverity.cs

```
namespace EmergencyMonitoring.Shared.Models;

public enum AlertSeverity
{
    Normal = 0,
    Warning = 1,
    Critical = 2,
    Emergency = 3
}
```

SensorData.cs

```
namespace EmergencyMonitoring.Shared.Models;

public class SensorData
{
    public int Id { get; set; }

    public string SensorId { get; set; } = string.Empty;

    public string Zone { get; set; } = string.Empty;

    public double Temperature { get; set; }

    public double SmokeLevel { get; set; }

    public double Vibration { get; set; }

    public DateTime Timestamp { get; set; }
}
```

ZoneStatus.cs

```
namespace EmergencyMonitoring.Shared.Models;

public class ZoneStatus
{
    public string Zone { get; set; } =
        string.Empty;

    public string Status { get; set; } =
        "Normal";
}
```

WPF

Services

ApiService.cs

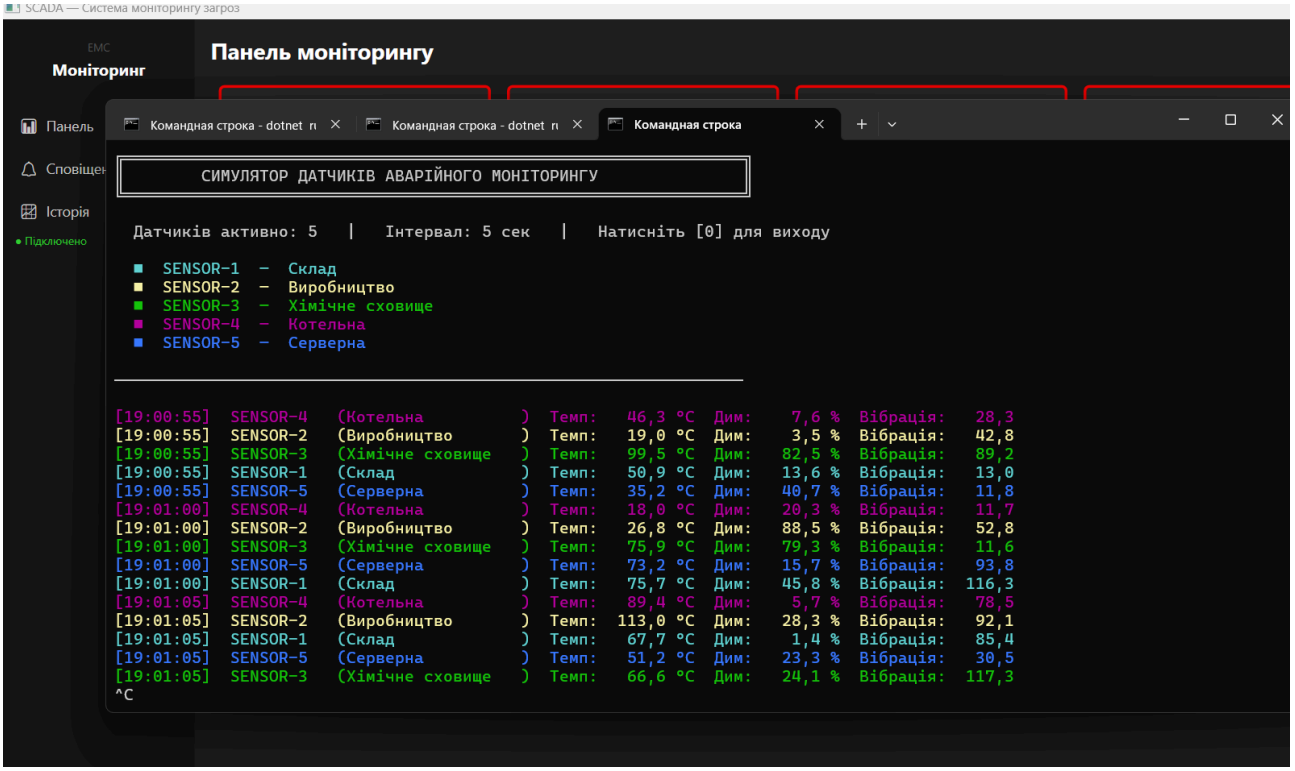
```
using EmergencyMonitoring.Shared.Models;
using System.Net.Http;
using System.Net.Http.Json;

namespace EmergencyMonitoring.WPF.Services;

public class ApiService
{
    private readonly HttpClient _http = new()
    {
        BaseAddress = new Uri("http://localhost:5223")
    };

    public async Task<List<SensorData>> GetHistoryAsync(string sensorId, int limit = 50)
    {
        var result = await _http.GetFromJsonAsync<List<SensorData>>(
            $"/api/sensors/history/{sensorId}?limit={limit}");
        return result ?? [];
    }

    public async Task<List<Alert>> GetAlertsAsync(bool? acknowledged = null)
    {
        var url = acknowledged.HasValue
            ? $"/api/alerts?acknowledged={acknowledged.Value.ToString().ToLower()}"
            : "/api/alerts";
        var result = await _http.GetFromJsonAsync<List<Alert>>(url);
        return result ?? [];
    }

    public async Task AcknowledgeAlertAsync(int id)
    {
        await _http.PostAsync($"/api/alerts/{id}/acknowledge", null);
    }
}
```

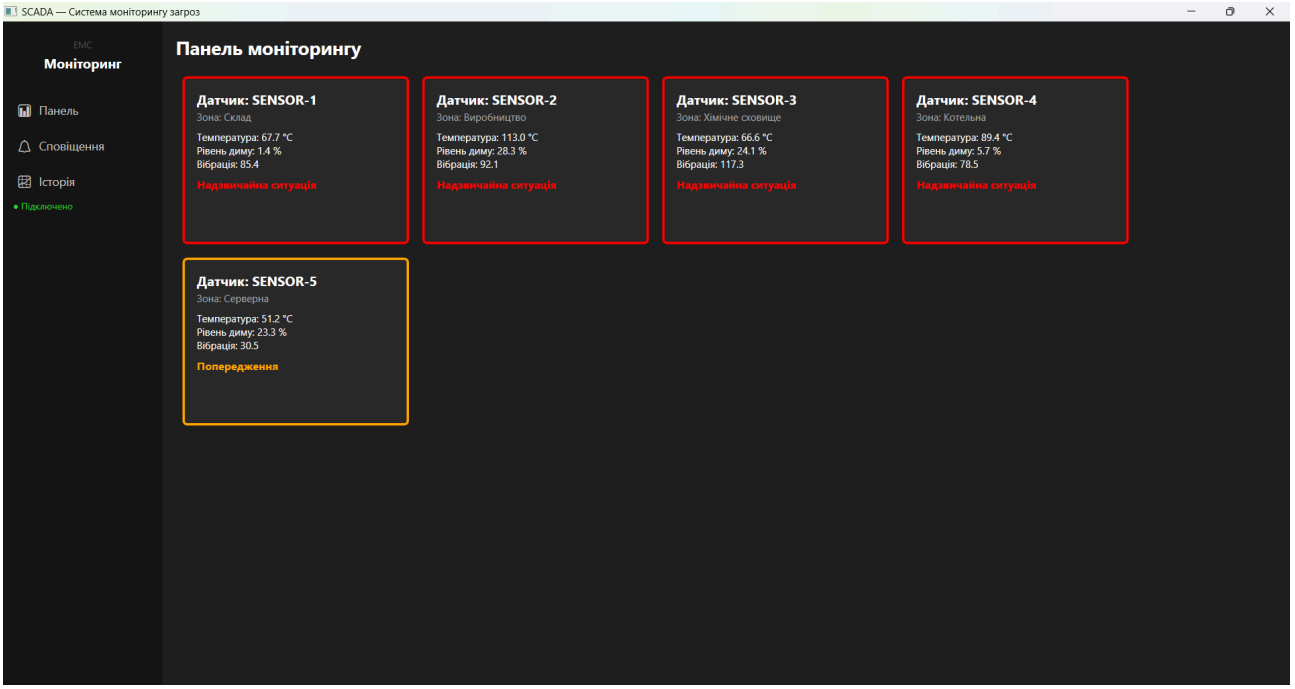
SignalRService.cs

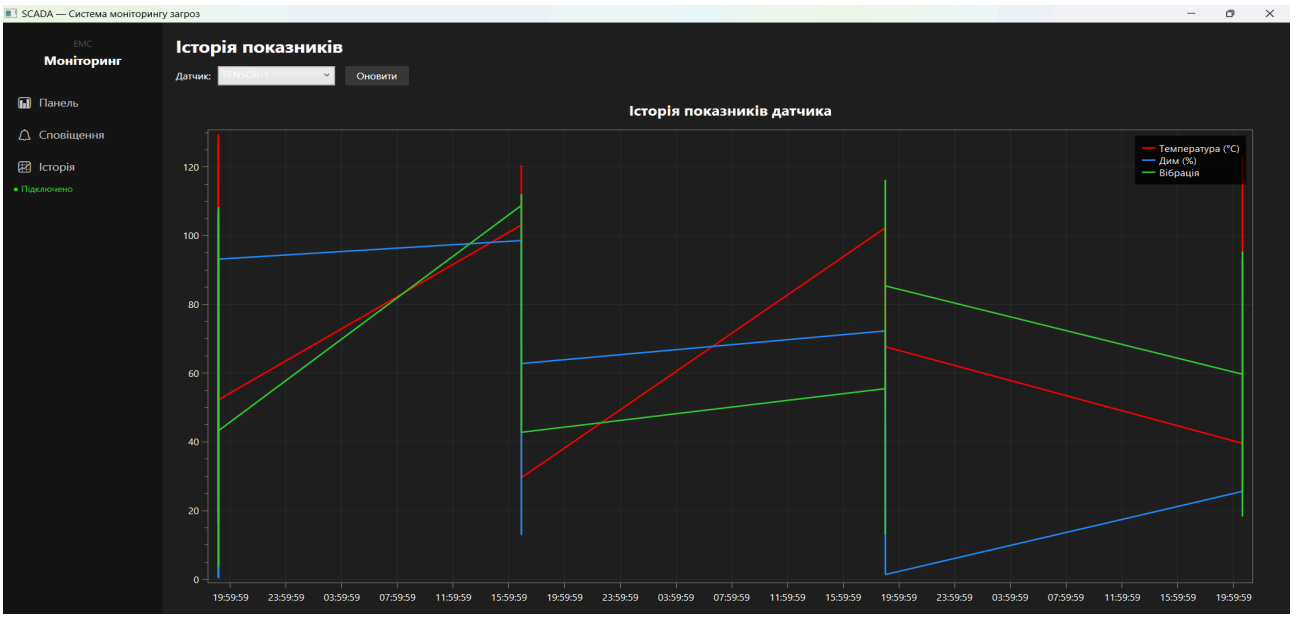
```
using EmergencyMonitoring.Shared.Models;
using Microsoft.AspNetCore.SignalR.Client;

namespace EmergencyMonitoring.WPF.Services;

public class SignalRService
{
    private HubConnection? _connection;

    public event Action<SensorData>? OnSensorData;
    public event Action<Alert>? OnAlert;
    public event Action<int>? OnAlertAcknowledged;

    public async Task StartAsync()
    {
        _connection = new HubConnectionBuilder()
            .WithUrl("http://localhost:5223/monitoringHub")
            .WithAutomaticReconnect()
            .Build();

        _connection.On<SensorData>("ReceiveSensorData", data =>
OnSensorData?.Invoke(data));
        _connection.On<Alert>("ReceiveAlert", alert => OnAlert?.Invoke(alert));
        _connection.On<int>("AlertAcknowledged", id => OnAlertAcknowledged?.Invoke(id));

        await _connection.StartAsync();
```

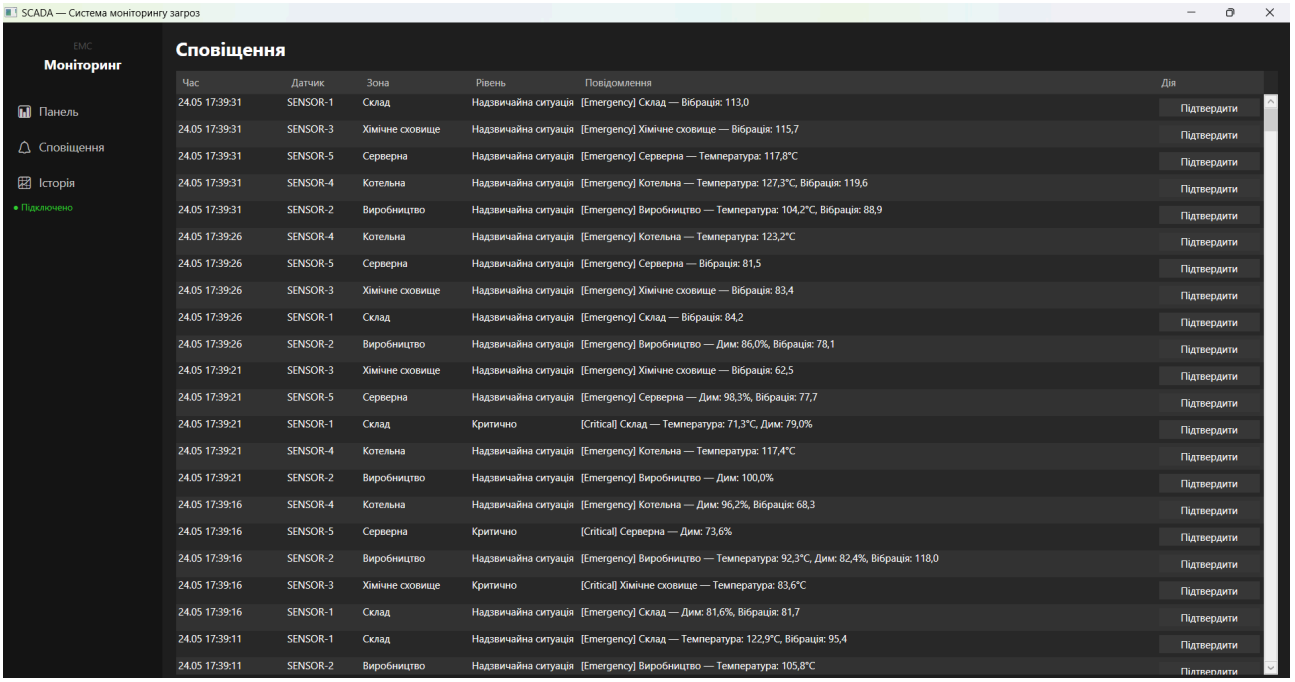
```
    }

    public async Task AcknowledgeAlertAsync(int alertId)
    {
        if (_connection?.State == HubConnectionState.Connected)
            await _connection.InvokeAsync("AcknowledgeAlert", alertId);
    }
}
```

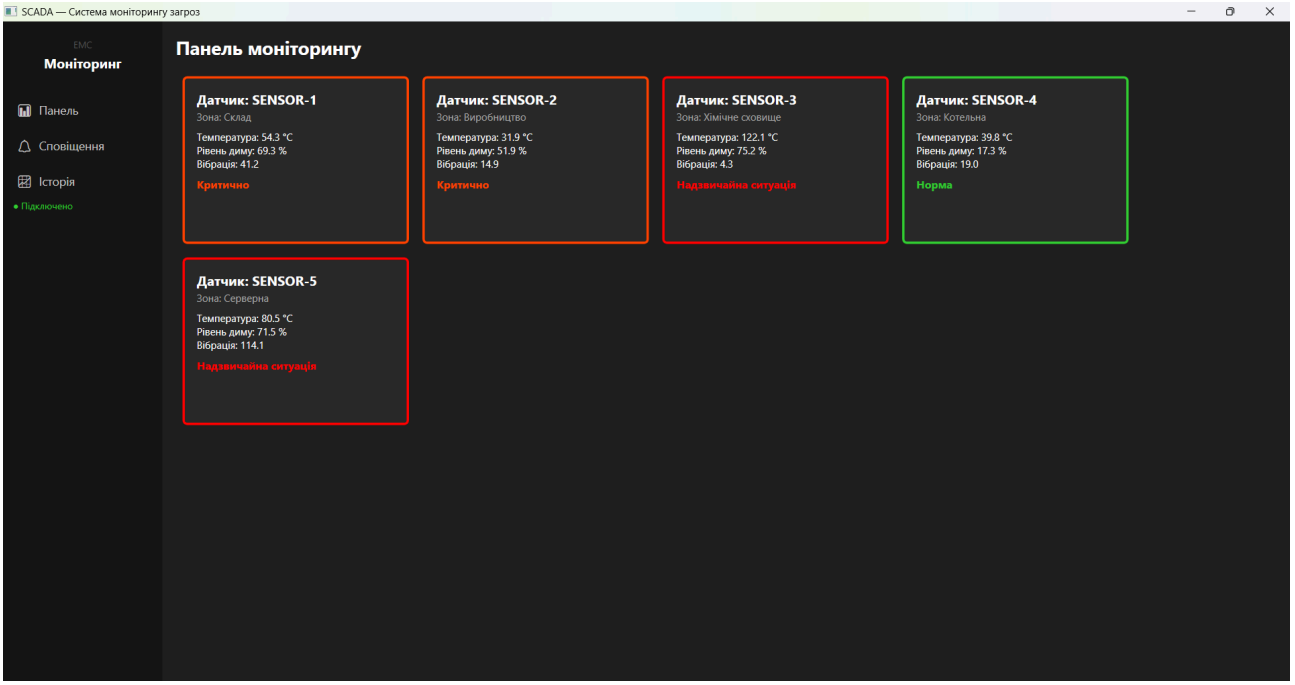
Додаток В






SCADA — Система моніторингу загроз
EMC
Моніторинг
Панель
Сповіщення
Історія
Підключено
Історія показників
Датчик:
SENSOR-1
Оновити
Історія показників датчика
Температура (°C)
Дим (%)
Вібрація
120
100
80
60
40
20
0
19:59:59
23:59:59
03:59:59
07:59:59
11:59:59
15:59:59
19:59:59
23:59:59
03:59:59
07:59:59
11:59:59
15:59:59
19:59:59
23:59:59
03:59:59
07:59:59
11:59:59
15:59:59
19:59:59


SCADA — Система моніторингу загроз
EMC
Моніторинг
Панель
Сповіщення
Історія
Підключено
Сповіщення
Час Датчик Зона Рівень Повідомлення Дія
24.05 17:39:31 SENSOR-1 Склад Надзвичайна ситуація [Emergency] Склад — Вібрація: 113,0 Підтвердити
24.05 17:39:31 SENSOR-3 Хімічне сховище Надзвичайна ситуація [Emergency] Хімічне сховище — Вібрація: 115,7 Підтвердити
24.05 17:39:31 SENSOR-5 Серверна Надзвичайна ситуація [Emergency] Серверна — Температура: 117,8°C Підтвердити
24.05 17:39:31 SENSOR-4 Котельна Надзвичайна ситуація [Emergency] Котельна — Температура: 127,3°C, Вібрація: 119,6 Підтвердити
24.05 17:39:31 SENSOR-2 Виробництво Надзвичайна ситуація [Emergency] Виробництво — Температура: 104,2°C, Вібрація: 88,9 Підтвердити
24.05 17:39:26 SENSOR-4 Котельна Надзвичайна ситуація [Emergency] Котельна — Температура: 123,2°C Підтвердити
24.05 17:39:26 SENSOR-5 Серверна Надзвичайна ситуація [Emergency] Серверна — Вібрація: 81,5 Підтвердити
24.05 17:39:26 SENSOR-3 Хімічне сховище Надзвичайна ситуація [Emergency] Хімічне сховище — Вібрація: 83,4 Підтвердити
24.05 17:39:26 SENSOR-1 Склад Надзвичайна ситуація [Emergency] Склад — Вібрація: 84,2 Підтвердити
24.05 17:39:26 SENSOR-2 Виробництво Надзвичайна ситуація [Emergency] Виробництво — Дим: 86,0%, Вібрація: 78,1 Підтвердити
24.05 17:39:21 SENSOR-3 Хімічне сховище Надзвичайна ситуація [Emergency] Хімічне сховище — Вібрація: 62,5 Підтвердити
24.05 17:39:21 SENSOR-5 Серверна Надзвичайна ситуація [Emergency] Серверна — Дим: 98,3%, Вібрація: 77,7 Підтвердити
24.05 17:39:21 SENSOR-1 Склад Критично [Critical] Склад — Температура: 71,3°C, Дим: 79,0% Підтвердити
24.05 17:39:21 SENSOR-4 Котельна Надзвичайна ситуація [Emergency] Котельна — Температура: 117,4°C Підтвердити
24.05 17:39:21 SENSOR-2 Виробництво Надзвичайна ситуація [Emergency] Виробництво — Дим: 100,0% Підтвердити
24.05 17:39:16 SENSOR-4 Котельна Надзвичайна ситуація [Emergency] Котельна — Дим: 96,2%, Вібрація: 68,3 Підтвердити
24.05 17:39:16 SENSOR-5 Серверна Критично [Critical] Серверна — Дим: 73,6% Підтвердити
24.05 17:39:16 SENSOR-2 Виробництво Надзвичайна ситуація [Emergency] Виробництво — Температура: 92,3°C, Дим: 82,4%, Вібрація: 118,0 Підтвердити
24.05 17:39:16 SENSOR-3 Хімічне сховище Критично [Critical] Хімічне сховище — Температура: 83,6°C Підтвердити
24.05 17:39:16 SENSOR-1 Склад Надзвичайна ситуація [Emergency] Склад — Дим: 81,6%, Вібрація: 81,7 Підтвердити
24.05 17:39:11 SENSOR-1 Склад Надзвичайна ситуація [Emergency] Склад — Температура: 122,9°C, Вібрація: 95,4 Підтвердити
24.05 17:39:11 SENSOR-2 Виробництво Надзвичайна ситуація [Emergency] Виробництво — Температура: 105,8°C Підтвердити


SCADA — Система моніторингу загроз
EMC
Моніторинг
Панель
Сповіщення
Історія
Підключено
Панель моніторингу
Датчик: SENSOR-1
Зона: Склад
Температура: 54.3 °C
Рівень диму: 69.3 %
Вібрація: 41.2
Критично
Датчик: SENSOR-2
Зона: Виробництво
Температура: 31.9 °C
Рівень диму: 51.9 %
Вібрація: 14.9
Критично
Датчик: SENSOR-3
Зона: Хімічне сховище
Температура: 122.1 °C
Рівень диму: 75.2 %
Вібрація: 4.3
Надзвичайна ситуація
Датчик: SENSOR-4
Зона: Котельна
Температура: 39.8 °C
Рівень диму: 17.3 %
Вібрація: 19.0
Норма
Датчик: SENSOR-5
Зона: Серверна
Температура: 80.5 °C
Рівень диму: 71.5 %
Вібрація: 114.1
Надзвичайна ситуація