НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
«___» _____2026р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою «Цифрові технології в енергетиці»
Спеціальність 122 «Комп'ютерні науки»
на тему: «Клієнтська частина вебсистеми моніторингу доступності програмних застосунків»

Виконав: студент 4 курсу, групи ТР-23
Пузенко Артем Андрійович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник: асистент Антон ПАСІЧНЮК _____
(посада, науковий ступінь, вчене звання, ім'я, ПРІЗВИЩЕ) (підпис)

Рецензент: зав. каф. ТАЕ, к.т.н., доц. Віталій ПЕШКО
(посада, науковий ступінь, вчене звання, ім'я, ПРІЗВИЩЕ) (підпис)

Н.контроль: ас. Артем МЕЛЬНИЧЕНКО _____
(посада, ім'я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
"___"_____2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

Пузенку Артему Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи : «Клієнтська частина вебсистеми моніторингу доступності програмних застосунків»
Науковий керівник Пасічнюк Антон Олексійович, асистент кафедри ЦТЕ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «28» травня 2026 року № 1992-с
2. Термін подання студентом роботи 08.06.2026
3. Вихідні дані до роботи мова програмування TypeScript, бібліотека React 19, інструмент збірки Vite, бібліотека керування станом Redux Toolkit, HTTP-клієнт Axios, маршрутизація React Router DOM, бібліотека форм і валідації React Hook Form, технологія real-time сповіщень SignalR, бібліотека візуалізації Recharts, система інтернаціоналізації i18next, мови розмітки та стилізації HTML, CSS, SCSS

Modules, Tailwind CSS і Bulma, взаємодія з REST API та GraphQL API, Web Worker для обчислення статистики, середовище розробки Visual Studio Code.

4. Перелік питань, які потрібно розробити

1) проаналізувати рішення для моніторингу доступності вебсервісів;

2) розглянути підходи до моніторингу та обґрунтувати вибір активного синтетичного методу;

3) визначити вимоги до клієнтської частини NeverDown;

4) спроєктувати архітектуру клієнтської частини системи;

5) реалізувати вебінтерфейс для роботи з моніторами, інцидентами, статистикою, сповіщеннями, профілем і локалізацією.

5. Орієнтований перелік ілюстративного матеріалу схема архітектури клієнтської частини системи, діаграма варіантів використання, діаграма зв'язків між модулями, структура даних/БД, схема алгоритму активного моніторингу, скріншоти вебінтерфейсу системи NeverDown

6. Дата видачі завдання 13.09.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	14.12.2025	
2.	Аналіз методів та засобів розв'язання задачі	23.03.2026-19.04.2026	
3.	Розробка архітектури та загальної структури системи	20.04.2026-26.04.2026	
4.	Розробка окремих підсистем	27.04.2026-03.05.2026	
5.	Програмна реалізація системи	04.05.2026-17.05.2026	
6.	Оформлення пояснювальної записки	18.05.2026-30.05.2026	
7.	Захист програмного забезпечення	13.05.2026	
8.	Передзахист	03.06.2025	
9.	Захист	16.06.2025-20.06.2025	

Студент _____ Артем ПУЗЕНКО
(підпис) (прізвище та ініціали)

Керівник _____ Антон ПАСІЧНЮК
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота викладена на 68 сторінках, містить 18 рисунків, 1 додаток та 12 джерел у списку використаної літератури.

Метою роботи є розроблення клієнтської частини вебсистеми моніторингу доступності програмних застосунків NeverDown, яка забезпечує керування моніторами, перегляд стану вебсервісів, аналіз статистики перевірок, роботу з інцидентами та отримання сповіщень у режимі реального часу. Об’єктом дослідження є процеси моніторингу доступності вебсервісів. Предметом дослідження є методи, підходи та програмні засоби побудови клієнтської частини вебсистеми моніторингу.

У роботі проаналізовано існуючі рішення для моніторингу доступності, розглянуто основні підходи до моніторингу вебсервісів та обґрунтовано вибір активного синтетичного моніторингу. Розроблено клієнтську частину системи на базі React 19, TypeScript, Redux Toolkit, Axios, SignalR, Recharts, i18next та Vite. Реалізовано модулі автентифікації, керування моніторами, перегляду деталізації монітора, обробки інцидентів, налаштування правил сповіщень, центру сповіщень, імпорту й експорту CSV, редагування профілю користувача та перемикання мови інтерфейсу.

Ключові слова: моніторинг доступності, вебсистема, клієнтська частина, React, TypeScript, Redux Toolkit, SignalR, REST API, GraphQL, SPA.

ANNOTATION

The thesis consists of 68 pages, contains 18 figures, 1 appendix, and 12 references.

The aim of the work is to develop the client-side of the NeverDown web system for monitoring the availability of software applications. The system provides monitor management, web service status tracking, statistical analysis of checks, incident handling, and real-time notifications. The object of the research is the process of monitoring web service availability. The subject of the research is the methods, approaches, and software tools used to build the client-side of an availability monitoring web system.

The thesis analyzes existing availability monitoring solutions, considers the main approaches to web service monitoring, and justifies the choice of active synthetic monitoring. The client-side of the system was developed using React 19, TypeScript, Redux Toolkit, Axios, SignalR, Recharts, i18next, and Vite. The implemented modules include authentication, monitor management, monitor detail view, incident handling, alert rule configuration, notification center, CSV import and export, user profile editing, and interface language switching.

Keywords: availability monitoring, web system, client-side, React, TypeScript, Redux Toolkit, SignalR, REST API, GraphQL, SPA.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

SPA — односторінковий вебзастосунок, у якому основна навігація виконується без повного перезавантаження сторінки.

API — програмний інтерфейс взаємодії між клієнтською та серверною частинами системи.

REST — архітектурний підхід до побудови вебінтерфейсів обміну даними.

GraphQL — мова запитів до API, яка дозволяє клієнту визначати потрібну структуру відповіді.

JWT — компактний токен для передавання відомостей про автентифікованого користувача.

URL — уніфікований покажчик ресурсу в мережі.

UI — користувацький інтерфейс програмної системи.

UX — користувацький досвід взаємодії з програмною системою.

CSV — текстовий формат подання табличних даних.

DOM — об’єктна модель документа, з якою взаємодіє браузер.

SignalR — технологія організації обміну подіями між сервером і клієнтом у реальному часі.

Web Worker — механізм браузера для виконання обчислень у фоновому потоці.

i18n — скорочене позначення інтернаціоналізації та перекладу інтерфейсу.

ВСТУП

Сучасний рівень цифровізації бізнесу, освіти, державних сервісів та інших сфер зробив стабільну роботу вебзастосунків одним із ключових показників якості програмного забезпечення. Періоди, коли сервіс недоступний недоступності сервісу може призвести до втрати користувачів, зниження довіри до системи, фінансових збитків або порушення внутрішніх бізнес-процесів. Саме тому своєчасне виявлення збоїв, контроль стану вебсервісів і оперативне інформування відповідальних осіб є важливими задачами для сучасних інформаційних систем.

Для вирішення цих проблем використовуються системи моніторингу доступності, які виконують регулярні перевірки вебресурсів, фіксують час відповіді, аналізують статус сервісів і формують інциденти у разі виявлення проблем. Важливу роль у таких системах відіграє клієнтська частина, оскільки саме через неї користувач керує моніторами, переглядає статистику, аналізує історію перевірок, налаштовує правила сповіщень та реагує на інциденти.

Ця робота присвячена розробці клієнтської частини вебсистеми моніторингу доступності програмних застосунків NeverDown. Система реалізована у вигляді сучасного SPA-застосунку з використанням React 19, TypeScript, Redux Toolkit, Axios, SignalR, Recharts та інших інструментів клієнтської розробки. Розроблений інтерфейс забезпечує створення та керування моніторами, перегляд стану сервісів, аналіз статистичних показників, роботу з інцидентами, отримання сповіщень у реальному часі та підтримку української й англійської мов інтерфейсу.

У межах роботи було проаналізовано підходи до моніторингу вебсервісів, розглянуто наявні рішення, обґрунтовано вибір активного синтетичного моніторингу, визначено технологічний стек клієнтської частини та продемонстровано роботу основних функцій системи NeverDown.

1 ЗАДАЧА ОНЛАЙН-МОНІТОРИНГУ ВЕБСЕРВІСІВ

Надання користувачу зрозумілої інформації про стан контрольованих сервісів є важливою задачею систем моніторингу. Клієнтська частина повинна не лише відображати поточний статус, а й допомагати користувачу зрозуміти контекст проблеми: коли вона виникла, як довго тривала, чи повторювалася раніше та які дії були виконані для її усунення.

1.1 Проблематика моніторингу доступності в сучасних системах

Моніторинг доступності є базовим елементом супроводу вебсервісів. У готових платформах часто добре реалізовано саму перевірку сервісу, але клієнтський інтерфейс може бути обмеженим для навчального або внутрішнього використання: частина функцій доступна лише у платних тарифах, інтеграція з власною серверною частиною є складною, а можливості локалізації або налаштування сценаріїв інцидентів не завжди достатні.

Для користувача важливо бачити не тільки факт недоступності, а й пов’язану з ним історію подій. Якщо інтерфейс показує лише останній стан сервісу, користувач не може оцінити тривалість простою, стабільність ресурсу та динаміку зміни показників. Саме тому клієнтська частина повинна поєднувати список ресурсів, сторінку деталізації, історію перевірок, статистичні показники та засоби роботи з інцидентами.

Окремою проблемою є організація сповіщень. У різних сценаріях користувач може потребувати повідомлення у застосунку, листа або інтеграції із зовнішнім каналом. Клієнтська частина має надавати зручний інтерфейс для керування такими правилами, не прив’язуючи опис вимог до конкретної технічної реалізації.

1.2 Формулювання задачі розробки клієнтської частини

Метою дипломної роботи є розробка клієнтської частини вебсистеми моніторингу доступності програмних застосунків NeverDown, яка забезпечує користувачу зручну роботу з моніторами, статистикою, інцидентами, профілем і правилами сповіщень:

- проаналізувати існуючі рішення для моніторингу доступності вебсервісів та визначити їхні сильні й слабкі сторони з погляду користувацького інтерфейсу;
- розглянути основні методи моніторингу вебсервісів і обґрунтувати підхід, що відповідає призначенню системи NeverDown;
- сформулювати функціональні та нефункціональні вимоги до клієнтської частини системи;
- спроєктувати архітектуру клієнтської частини з урахуванням маршрутизації, централізованого стану, взаємодії з API, обробки подій у реальному часі та відображення статистики;
- реалізувати вебінтерфейс, який дає користувачу можливість автентифікуватися, керувати моніторами, переглядати статистику, працювати з інцидентами, налаштовувати сповіщення та змінювати параметри профілю;
- продемонструвати роботу основних сценаріїв системи моніторингу.

1.3 Порівняльний аналіз існуючих рішень для моніторингу вебсервісів

Існує багато готових рішень для моніторингу доступності вебсервісів, які відрізняються типами перевірок, способом подання статистики, підтримкою інцидентів, каналами сповіщень та умовами використання. Їхній аналіз дає змогу

визначити сильні сторони наявних платформ і сформувати вимоги до власної системи NeverDown.

Сучасні системи моніторингу доступності переважно реалізуються у вигляді вебінтерфейсів або односторінкових застосунків. Зазвичай вони містять дашборд із переліком моніторів, сторінки деталізації з графіками, журнал перевірок, центр інцидентів та налаштування сповіщень. Такий підхід дозволяє користувачу швидко оцінити стан сервісів і перейти до аналізу проблеми.

Одним із популярних рішень є UptimeRobot. Сервіс має простий інтерфейс, підтримує базові перевірки доступності та надає безкоштовний тариф для невеликої кількості моніторів. До його переваг можна віднести простоту використання, швидке створення моніторів і зрозуміле відображення статусів. Водночас можливості гнучкого керування інцидентами, коментування та налаштування сповіщень є обмеженими.

Pingdom є більш професійним рішенням, орієнтованим на корпоративних користувачів. Воно підтримує synthetic monitoring, real user monitoring, status pages та складні сценарії перевірок. Перевагою Pingdom є широкий набір аналітичних інструментів і детальні графіки продуктивності. Недоліками можна вважати високу вартість і надмірну складність для невеликих команд або навчальних проєктів.

Better Uptime, що входить до екосистеми Better Stack, робить акцент на інцидент-менеджменті, on-call-процесах та інтеграціях із зовнішніми сервісами. Його інтерфейс є сучасним і зручним, а робота з інцидентами добре структурована. До переваг належать підтримка acknowledge/resolve-сценаріїв, коментарів та інтеграцій зі сповіщеннями. Серед недоліків можна виділити обмеження безкоштовного тарифу та орієнтацію на ширшу DevOps-інфраструктуру.

Отже, аналіз існуючих рішень показав, що більшість платформ мають сильні сторони у сфері перевірки доступності.

Під час порівняння готових платформ було враховано не лише наявність базових перевірок, а й зручність роботи з інцидентами, прозорість історії подій, можливість локалізації інтерфейсу та гнучкість інтеграції з власною серверною

частиною. Для дипломного проєкту особливо важливо, щоб клієнтська частина не була прив’язана до закритої екосистеми постачальника і могла розвиватися разом із серверним API.

UptimeRobot є прикладом простого сервісу для регулярних перевірок, але не охоплює повного керування інцидентами. Pingdom і Better Uptime мають розвинуті можливості для командної роботи, однак можуть бути надлишковими для навчального або внутрішнього проєкту. Це підтверджує доцільність створення власної системи, орієнтованої на ключові сценарії.

NeverDown має забезпечувати створення монітора, перегляд його стану і статистики, отримання сповіщень, роботу з інцидентами та налаштування правил повідомлень. Ці функції покривають основні потреби користувача й залишають можливість подальшого розширення системи.

Порівняння існуючих рішень наведено в таблиці 1.1.

Таблиця 1.1 – Порівняння існуючих рішень за основними критеріями

Критерій	**UptimeRobot**	**Pingdom**	**Better Uptime**	**NeverDown**
Робота з моніторами	Базове створення і перегляд	Розширені сценарії	Зручне керування	Створення, редагування, призупинення, відновлення, імпорт та експорт
Інциденти	Обмежена модель	Є аналітика подій	Сильний incident-flow	Підтвердження, розв’язання та коментарі
Локалізація інтерфейсу	Обмежена	Обмежена	Залежить від платформи	Українська та англійська мови
Гнучкість інтеграції	Закрита платформа	Комерційна екосистема	Екосистема Better Stack	Орієнтація на власну серверну частину

1.4 Загальні вимоги до клієнтської частини

Після аналізу предметної області та існуючих рішень визначено, що клієнтська частина системи має забезпечувати повний і зрозумілий цикл роботи користувача з результатами моніторингу доступності.

Система має надавати можливість реєстрації, входу до захищеної частини та завершення сеансу роботи. Доступ до робочих сторінок має бути обмежений для неавтентифікованих користувачів.

Користувач має мати можливість створювати, редагувати, видаляти, тимчасово призупиняти та відновлювати монітори, а також переглядати їхній поточний стан у списку та на сторінці деталізації.

Інтерфейс має відображати історію перевірок, ключові статистичні показники, періоди недоступності та графічну динаміку зміни показників, щоб користувач міг швидко оцінити стан сервісу.

Система має підтримувати роботу з інцидентами: перегляд переліку, деталізацію, підтвердження, розв’язання, коментування та збереження контексту прийнятих рішень.

Користувач має мати можливість налаштовувати правила отримання сповіщень і бачити оперативні повідомлення про важливі зміни стану сервісів.

Клієнтська частина має підтримувати редагування профілю користувача, зміну персональних даних, аватара, пароля та мови інтерфейсу.

Нефункціональні вимоги включають модульність архітектури, зрозумілу маршрутизацію, типізовані моделі даних, централізоване керування станом, обробку помилок, адаптивність інтерфейсу та можливість подальшого розширення.

Діаграму варіантів використання вебсистеми NeverDown наведено на рисунку 1.1.

Рисунок 1.1 – Діаграма варіантів використання вебсистеми NeverDown

З діаграми видно, що неавторизований користувач взаємодіє лише зі сценаріями входу та реєстрації, а основні функції моніторингу доступні після автентифікації. Такий поділ відповідає вимогам безпеки та визначає межі відповідальності клієнтської частини.

2 МЕТОДИ ТА ПІДХОДИ ДО МОНІТОРИНГУ ВЕБСЕРВІСІВ

У цьому розділі розглянуто основні методи моніторингу вебсервісів, які застосовуються для виявлення недоступності, деградації продуктивності та технічних збоїв. Для дипломної системи важливо не лише перелічити наявні підходи, а й визначити, який із них найкраще відповідає задачі регулярної перевірки доступності зовнішніх сервісів і своєчасного формування інцидентів.

2.1 Класифікація підходів до моніторингу вебсервісів

Підходи до моніторингу вебсервісів можна класифікувати за джерелом даних і способом виявлення проблеми. Активні методи самостійно ініціюють перевірки з боку системи моніторингу, пасивні методи аналізують поведінку реальних користувачів, агентні методи збирають метрики безпосередньо з інфраструктури, а журнальні методи працюють із подіями та записами виконання застосунку. Кожен підхід вирішує окрему задачу, тому вибір методу залежить від того, чи потрібно контролювати зовнішню доступність сервісу, внутрішній стан серверів або реальний користувацький досвід.

2.1.1 Активний синтетичний моніторинг

Активний синтетичний моніторинг передбачає ініціативне виконання перевірок за наперед заданим розкладом. Система сама формує перевірку, отримує результат, фіксує час відповіді та визначає, чи потрібно створювати інцидент або сповіщення.

Перевага підходу полягає в незалежності від реального користувацького трафіку. Проблема може бути виявлена ще до того, як користувач спробує скористатися сервісом. Для клієнтської частини це означає потребу у відображенні поточного статусу, історії перевірок, статистики доступності та періодів простою.

Схему активного синтетичного моніторингу наведено на рисунку 2.1.

Рисунок 2.1 – Потік активного синтетичного моніторингу

Як видно з рисунка 2.1, активний синтетичний моніторинг має послідовний цикл роботи: система ініціює перевірку, отримує відповідь від цільового сервісу, зберігає результат і на основі цього результату оновлює стан монітора. Якщо перевірка завершується помилкою або показники виходять за допустимі межі, система може сформувати інцидент і повідомити користувача. Для клієнтської частини важливо коректно відобразити всі етапи цього циклу: поточний статус, час останньої перевірки, історію змін, статистику доступності та повідомлення про проблеми.

2.1.2 Пасивний моніторинг реальних користувачів

Пасивний моніторинг реальних користувачів збирає дані під час фактичних сесій у браузері або клієнтському застосунку. Він добре відображає реальний

користувацький досвід, оскільки враховує пристрій, мережеві умови, географію користувача та помилки, що виникають під час взаємодії з інтерфейсом.

Обмеження підходу полягає в тому, що дані з’являються лише за наявності активних користувачів. Якщо сервіс недоступний у період низького трафіку, пасивний моніторинг може виявити проблему із запізненням. Схему такого підходу наведено на рисунку 2.2.

Рисунок 2.2 – Потік пасивного моніторингу реальних користувачів

З рисунка 2.2 видно, що пасивний моніторинг залежить від активності користувачів, оскільки дані надходять під час реальної роботи із застосунком. Він корисний для аналізу користувацького досвіду, продуктивності інтерфейсу та помилок у браузері. Однак за відсутності трафіку такий підхід може не виявити проблему вчасно, тому не замінює активний моніторинг доступності.

2.1.3 Агентний та інфраструктурний моніторинг

Агентний моніторинг передбачає встановлення програмного агента на сервер, контейнер або вузол інфраструктури. Він у фоновому режимі збирає внутрішні метрики: використання процесора, пам’яті, диска, мережі, а також стан процесів і служб. Такий підхід надає детальну інформацію про роботу системи та допомагає аналізувати продуктивність і причини інцидентів. Водночас він

потребує адміністративного доступу, додаткового налаштування та є складнішим у розгортанні.

2.1.4 Журнальний та подієвий моніторинг

Журнальний та подієвий моніторинг ґрунтується на збиранні логів, винятків і системних подій, що допомагає діагностувати причини збоїв. Однак він не завжди відображає доступність сервісу для користувача, тому його доцільно поєднувати з активними перевірками.

2.2 Порівняльна характеристика методів моніторингу

Методи моніторингу доцільно порівняти за джерелом даних, складністю впровадження, швидкістю виявлення проблем і можливістю формування інцидентів. Порівняння основних підходів наведено в таблиці 2.1.

Таблиця 2.1 – Порівняння підходів до моніторингу вебсервісів

Підхід	**Що вимірює**	**Як збираються дані**	**Переваги**	**Обмеження**
Активний синтетичний моніторинг	Доступність, час відповіді, код відповіді, факт збою	Система сама періодично виконує HTTP/HTTPS/TCP/PING-перевірки	Виявляє проблему до звернення користувача; не потребує встановлення агента; добре підходить для інцидентів і сповіщень	Показує зовнішню доступність, але не пояснює всі внутрішні причини збою

Кінець таблиці 2.1

Пасивний моніторинг реальних користувачів	Швидкість завантаження, помилки браузера, поведінку реальних відвідувачів	Дані надходять із клієнтських браузерів під час реальних сесій	Добре відображає фактичний користувацький досвід	Не фіксує проблему до появи користувача; залежить від трафіку
Агентний інфраструктурний моніторинг	CPU, RAM, диск, процеси, мережеві метрики, стан сервера	На сервер або вузол інфраструктури встановлюється агент	Дає глибокі технічні метрики та корисний для пошуку першопричин	Потребує доступу до інфраструктури та складнішого розгортання
Журнальний та подієвий моніторинг	Події застосунку, записи журналів, винятки, помилки виконання	Система збирає та аналізує логи, події або повідомлення з сервісів	Корисний для діагностики причин інцидентів і аудиту подій	Не завжди показує зовнішню доступність сервісу з погляду користувача

З наведеного порівняння видно, що активний синтетичний моніторинг найкраще відповідає задачі системи NeverDown. Він дозволяє самостійно перевіряти доступність сервісів, швидко виявляти збої та формувати інциденти без залежності від реального користувацького трафіку.

2.3 Підходи клієнтської частини до відображення результатів моніторингу

Клієнтська частина системи моніторингу повинна перетворювати технічні результати перевірок на зрозумілу для користувача картину. Для цього доцільно поєднувати короткі статуси, числові метрики, графіки, таблицю історії та окремий розділ інцидентів.

Список моніторів має давати швидке уявлення про стан усіх ресурсів. Сторінка деталізації повинна пояснювати причини зміни статусу через історію перевірок, графік часу відповіді та агреговані показники. Центр сповіщень має доповнювати ці дані оперативними повідомленнями, щоб користувач не пропускав важливі події.

2.4 Обґрунтування вибраного підходу розроблюваної системи

Для системи NeverDown основним підходом обрано активний синтетичний моніторинг. Він відповідає задачі контролю доступності, оскільки дозволяє регулярно перевіряти ресурси, фіксувати результати, створювати інциденти та інформувати користувача незалежно від наявності реального трафіку.

Клієнтська частина при цьому повинна показувати не лише останній статус, а й історію, статистику, періоди простою та контекст інцидентів. Саме тому в NeverDown передбачені сторінки списку моніторів, деталізації монітора, інцидентів, сповіщень і профілю користувача.

Активний синтетичний моніторинг добре узгоджується з моделлю інцидентів, оскільки регулярні перевірки дозволяють виявляти стійку недоступність, а не одиничні помилки. Це зменшує кількість хибних сповіщень і дає змогу формувати інцидент при повторенні проблеми або перевищенні порогу.

Для клієнтської частини це означає відображення статусу, історії перевірок, простоїв і середнього часу відповіді. Тому інтерфейс містить сторінку деталізації монітора, графік, історію перевірок і блок простоїв для оцінки впливу збою.

Вибраний підхід є достатньо універсальним для навчального прототипу і водночас практичним для реального використання. Він не вимагає встановлення агентів у сторонню інфраструктуру, може працювати з різними типами ресурсів і дозволяє масштабувати систему шляхом додавання нових типів перевірок або нових правил обробки результатів.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ ЧАСТИНИ СИСТЕМИ

У цьому розділі описано програмну реалізацію клієнтської частини вебсистеми моніторингу доступності NeverDown. Основну увагу приділено архітектурі застосунку, розподілу відповідальності між модулями, маршрутизації, керуванню станом, взаємодії з серверною частиною, сповіщенням у реальному часі та відображенню статистики.

3.1 Архітектура клієнтської частини вебсистеми

Клієнтська частина NeverDown побудована як односторінковий вебзастосунок із модульною організацією. Архітектура розділяє відповідальність між шаром сторінок, повторно використовуваними компонентами, централізованим станом, API-модулями, сервісами реального часу, типізованими моделями предметної області та допоміжними механізмами обробки статистики.

Сторінки відповідають за композицію інтерфейсу та сценарії взаємодії користувача. Повторно використовувані компоненти реалізують картки моніторів, модальні вікна, повідомлення, центр сповіщень, форми та блоки деталізації. Такий поділ зменшує дублювання і дозволяє однаково відображати подібні об’єкти в різних частинах системи.

Централізований стан використовується для даних автентифікації, профілю користувача, списку моніторів і деталізації вибраного монітора. Це важливо для системи моніторингу, оскільки один і той самий об’єкт може бути потрібний на декількох сторінках: у списку, деталізації, правилах сповіщень та інцидентах.

API-модулі ізолюють взаємодію з серверною частиною. Компоненти інтерфейсу не формують запити безпосередньо, а працюють через дії стану або

спеціалізовані функції доступу до даних. Це спрощує обробку помилок, повторне використання запитів і подальшу зміну серверних контрактів.

Окремий сервіс реального часу відповідає за підключення до каналу сповіщень і доставку подій у компоненти інтерфейсу. Для обчислення агрегованої статистики використовується фоновий механізм, який не блокує основний потік інтерфейсу під час обробки історії перевірок.

Загальну архітектуру клієнтської частини наведено на рисунку 3.1.

Рисунок 3.1 – Архітектура клієнтської частини NeverDown

Маршрутизація застосунку відокремлює публічні сторінки від захищених сценаріїв. Публічна частина містить головну сторінку, вхід і реєстрацію. Захищена частина доступна після перевірки автентифікації та містить робочі сторінки моніторингу, інцидентів, сповіщень і профілю.

Схему маршрутизації компонентів наведено на рисунку 3.2.

Рисунок 3.2 – Схема маршрутизації компонентів клієнтської частини

Моделі предметної області визначають структуру даних, з якими працює клієнтська частина вебсистеми NeverDown. Вони забезпечують узгоджене представлення об’єктів у сторінках, компонентах, API-модулях і централізованому стані.

До ключових сутностей належать користувач, монітор, результат перевірки, агрегована статистика, інцидент, коментар до інциденту та правило сповіщення. Центральною сутністю є монітор, оскільки з ним пов’язані результати перевірок, статистика доступності, інциденти та сповіщення. Користувач керує власними моніторами, а інциденти й коментарі дають змогу фіксувати проблемні ситуації та процес їх опрацювання. Їхні зв’язки наведено на рисунку 3.3.

Рисунок 3.3 – Діаграма основних моделей даних клієнтської частини

Як видно з рисунка 3.3, центральною сутністю клієнтської частини є Monitor, оскільки саме навколо монітора групуються результати перевірок, агрегована статистика, правила сповіщень та інциденти. Користувач може мати декілька моніторів, а кожен монітор, у свою чергу, пов’язаний з історією перевірок і показниками доступності. Така структура дозволяє відображати як загальний список ресурсів, так і детальну інформацію про окремий сервіс.

Окрему роль відіграє модель Incident, яка фіксує період проблеми та стан її опрацювання. Коментарі до інциденту зберігають контекст дій користувача і допомагають відстежувати процес реагування. Модель Alert пов’язана з монітором і визначає спосіб інформування користувача про важливі зміни стану. Завдяки такому розподілу даних клієнтська частина може узгоджено відображати монітори, статистику, інциденти та сповіщення в різних розділах інтерфейсу.

3.2 Опис та обґрунтування інструментів розробки

Альтернативою TypeScript є використання звичайного JavaScript або JSDoc-анотацій, однак ці підходи слабше контролюють контракти між шарами застосунку. У системі моніторингу клієнтська частина постійно обмінюється структурованими даними з серверною частиною: моніторами, інцидентами, користувачами, правилами сповіщень. Статична типізація зменшує ризик помилок у назвах полів, статусах сутностей і параметрах API-запитів ще до запуску застосунку.

TypeScript - суворо типізована надбудова над JavaScript, що компілюється у звичайний JavaScript та виконується у будь-якому сучасному браузері [2]. Сувора типізація дозволяє виявляти більшість помилок ще на етапі компіляції, до того, як код потрапить у браузер.

У межах проєкту TypeScript використовується для:

- типізації моделей даних предметної області;

- параметризації Redux-слайсів та їхніх асинхронних `createAsyncThunk` через generic-типи;

- забезпечення безпечного автодоповнення у середовищі розробника та запобігання помилкам передачі неправильних даних між шарами;

- забезпечення типізації пропсів React-компонентів та форм react-hook-form через generic.

Конфігурація компілятора задана у файлі tsconfig.app.json із цільовим діапазоном ES2022.

Типізація моделей предметної області підвищує надійність клієнтської частини. Статуси моніторів, інцидентів і правил сповіщень мають обмежений набір значень, тому їх доцільно описувати через типи або перелічення. Це допомагає уникнути розбіжностей між компонентами, наприклад, коли один очікує Online, а інший передає online.

Також TypeScript корисний під час роботи з формами. Дані для створення монітора або редагування профілю перевіряються ще до надсилання на сервер. Поєднання типів, правил валідації та API-контрактів дозволяє виявляти помилки вже на етапі розробки.

3.3 Реалізація взаємодії з серверною частиною

Інструмент збирання відповідає за перетворення вихідного коду TypeScript, JSX, стилів і ресурсів у виконуваний пакет клієнтської частини. Для проєкту важливо забезпечити швидкий цикл розробки, оскільки інтерфейс системи містить значну кількість сторінок, форм і станів. Vite використовує нативні ES-модулі браузера під час розробки, тому не перебудовує весь застосунок після кожної зміни. Це скорочує час зворотного зв’язку та підвищує продуктивність розробки.

Vite обрано для клієнтської частини NeverDown через швидкий локальний запуск, гаряче оновлення сторінки та зручну конфігурацію. Плагін mkcert дозволяє тестувати сценарії, що потребують локального HTTPS.

Vite використовує esbuild для швидкої транспіляції TypeScript і Rollup для фінальної збірки з оптимізаціями, зокрема code splitting і tree shaking [4].

У межах проєкту Vite використовується для:

- запуску сервера розробки на порту 5173 з гарячим перезавантаженням;
- роботи з Tailwind CSS через офіційний плагін;
- автоматичної генерації самопідписаного SSL-сертифіката для HTTPS у режимі розробки через vite-plugin-mkcert - це необхідно для тестування каналу реального часу SignalR та інших HTTPS-залежних функцій.

Швидкість локальної розробки безпосередньо впливає на якість клієнтської частини. Vite забезпечує швидкий запуск і гаряче оновлення модулів, тому розробник може оперативно перевіряти зміни компонентів, стилів і різні стани інтерфейсу.

Промислова збірка Vite також оптимізує залежності, виконує поділ коду та мінімізує ресурси. Завдяки цьому застосунок швидше завантажується, а користувач швидше переходить до перегляду стану сервісів.

3.4 Реалізація керування станом застосунку

Керування станом потрібне, оскільки дані моніторів, інцидентів, сповіщень і автентифікації використовуються на різних сторінках. Централізоване сховище запобігає дублюванню запитів і неузгодженості даних. Redux Toolkit розділяє стан на зрізи та спрощує асинхронні операції, роблячи інтерфейс передбачуванішим.

Redux Toolkit обрано для контрольованого управління станом і відстеження змін інтерфейсу. Це важливо для NeverDown, оскільки система одночасно працює з REST-запитами, GraphQL-операціями та сповіщеннями через SignalR.

Він є рекомендованим підходом до використання Redux у React, зменшує кількість шаблонного коду та надає зручні інструменти configureStore, createSlice і createAsyncThunk [3].

У межах проєкту Redux Toolkit використовується для:

- централізованого зберігання стану автентифікації, профілю користувача, переліку моніторів та деталей конкретного монітора у файлі store.ts;

- організації асинхронних операцій через createAsyncThunk - кожен thunk має автоматично три стани (pending, fulfilled, rejected), що використовуються для відображення індикаторів завантаження та повідомлень про помилки;

- безпечної мутабельної логіки оновлення масивів та об’єкті;

- інтеграції з Redux DevTools для зручного налагодження з можливістю перегляду історії змін стану.

Загальну схему взаємодії React-компонентів, асинхронних дій та централізованого сховища стану подано на рисунку 3.1.

Рисунок 3.1 – Схема керування станом застосунку через Redux Toolkit

Наведена схема демонструє, що React-компоненти не змінюють стан напряму, а взаємодіють зі сховищем через дії та асинхронні thunk-функції. Це дозволяє централізовано контролювати оновлення даних, обробку помилок і стан завантаження. Завдяки такій організації логіка роботи з даними відокремлена від інтерфейсу, що спрощує супровід і подальше розширення клієнтської частини NeverDown.

У Redux-сховищі доцільно розділяти короткочасний стан інтерфейсу та предметні дані, отримані з API. Наприклад, відкриття модального вікна або значення локального поля пошуку може залишатися на рівні компонента, тоді як перелік моніторів, поточний користувач, деталі монітора і результат асинхронного запиту мають бути доступні різним сторінкам. Такий поділ допомагає уникати надмірного ускладнення глобального стану.

Асинхронні thunk-дії зручні тим, що кожна операція має передбачуваний життєвий цикл. Поки запит виконується, інтерфейс може показувати індикатор завантаження; після успішної відповіді дані оновлюються у сховищі; у випадку помилки користувач отримує зрозуміле повідомлення. Для системи моніторингу це важливо, оскільки запити до моніторів, інцидентів і правил сповіщень можуть виконуватися незалежно і не повинні блокувати весь застосунок.

3.5 Реалізація інтерфейсу та візуалізації даних

Для маршрутизації в NeverDown обрано React Router DOM, оскільки проєкт реалізовано як SPA-застосунок без серверного рендерингу. Бібліотека підтримує вкладені й захищені маршрути, параметри URL та програмні переходи, що дозволяє переміщатися між сторінками без повного перезавантаження [6].

У проєкті react-router-dom використовується для опису маршрутів у src/Root.tsx, рендерингу вкладених сторінок через Outlet, захисту приватних сторінок компонентом RequireAuth, отримання параметрів через useParams і навігації через useNavigate.

3.6 Реалізація сповіщень у реальному часі

HTTP-клієнт у NeverDown є єдиним шлюзом між інтерфейсом і серверною частиною. Він надсилає запити, додає токен доступу, обробляє помилки автентифікації, поновлює сесію та повторює запити після оновлення токена.

Централізація цієї логіки зменшує дублювання коду й забезпечує однакову поведінку всіх API-модулів.

Таблиця 3.4 – Порівняння HTTP-клієнтів для взаємодії з REST API

Критерій	**Axios**	**Fetch API**	**ky**
Перехоплювачі запитів і відповідей	Вбудований механізм interceptor	Потрібна власна обгортка	Є hooks, але менша поширеність у стеку React
Оновлення JWT-сесії	Зручно реалізується централізовано	Потребує ручної реалізації для кожного сценарію	Можливе через hooks, але складніше для черги запитів

Кінець таблиці 3.4

Обробка помилок	Єдина структура помилок і статусів	Потрібна ручна перевірка response.ok	Зручна обробка, але менш гнучка для складних interceptor-сценаріїв
Передача cookie	Підтримується через withCredentials	Підтримується через credentials	Підтримується через fetch-налаштування
Доцільність для NeverDown	Найкращий вибір для refresh-черги та централізованої авторизації	Занадто багато повторюваної логіки	Можливий, але не дає суттєвої переваги

Axios обрано через потребу в єдиній точці контролю авторизації та помилок. У системі моніторингу це зменшує ризик ситуацій, коли частина запитів виконується з простроченим токеном або обробляє помилки інакше, ніж решта застосунку.

Axios - повноцінна HTTP-бібліотека з потужним механізмом перехоплювачів запиту та відповіді [5]. На відміну від базового fetch API, Axios підтримує централізовану обробку авторизації, автоматичне поновлення токенів та зручну обробку помилок.

У межах проєкту Axios використовується для:

- усіх REST-запитів до серверної частини через єдиний екземпляр із базовим URL `https://newerdown.online`;

- автоматичного додавання заголовка авторизації через перехоплювач запиту;

- обробки помилки 401 у перехоплювачі відповіді: при спливанні access-токена клієнт автоматично надсилає refresh-запит, оновлює токен та повторно виконує початковий запит;

- черги поновлення токена для коректної обробки декількох паралельних 401-запитів - лише один refresh-запит виконується одночасно, інші чекають на новий токен;

Централізація HTTP-клієнта також спрощує дотримання однакових правил безпеки. Якщо токен доступу додається в одному перехоплювачі, окремі API-модулі не дублюють цю логіку і не можуть випадково надіслати захищений запит без авторизаційного заголовка. Аналогічно, єдина обробка помилки 401 дозволяє уникнути ситуації, коли різні сторінки по-різному реагують на завершення сесії.

Для користувача така логіка проявляється як непомітне продовження роботи після оновлення access-токена. Якщо під час перегляду моніторів або інцидентів токен втратив чинність, клієнтська частина спочатку намагається поновити його через refresh-механізм, а потім повторює початковий запит. Це підвищує зручність використання системи і зменшує кількість примусових виходів із сеансу.

- передачі HttpOnly-cookie з refresh-токеном через прапор withCredentials: true.

3.7 Бібліотека керування формами та їх валідацією

Альтернативами React Hook Form є Formik і повністю ручне керування станом полів через React. Для проєкту з великою кількістю форм ручний підхід збільшив би обсяг коду та кількість повторюваної валідації, а Formik частіше призводить до зайвих повторних рендерингів через контрольований стан полів. React Hook Form краще відповідає вимогам продуктивності, оскільки використовує неконтрольовані елементи форми та оновлює лише необхідні частини інтерфейсу.

React Hook Form - бібліотека для роботи з формами в React, що реалізує uncontrolled-підхід (значення зберігаються у нативних DOM-елементах) [7]. Це забезпечує дуже високу продуктивність навіть для складних форм та мінімальну кількість ререндерів.

У межах проєкту react-hook-form використовується для:

- декларативної реєстрації полів форм через метод register() із вбудованою валідацією;

- валідації email-полів через регулярний вираз;

- мінімальної довжини паролів (8 символів) та перевірки співпадіння поля Confirm Password з основним полем Password через опцію validate;

- складного регулярного виразу для URL/IP-адрес моніторів, що приймає `https://example.com`, 192.168.1.1, доменні імена з портами;

- валідації українських телефонних номерів за маскою у профілі користувача.

3.8 Підсистема перекладу інтерфейсу

Серед альтернатив інтернаціоналізації можна виділити FormatJS або власну систему словників, однак вони або мають іншу модель форматування, або потребують додаткового супроводу. i18next надає готовий механізм визначення мови, розділення перекладів на простори назв і збереження вибору користувача. Для системи, що підтримує українську та англійську мови, це зменшує обсяг власного коду та спрощує подальше додавання нових мов.

i18next - потужна бібліотека інтернаціоналізації з підтримкою множин, namespaces, плюралізації та підключення хмарних сервісів [8]. Має офіційну React-обгортку react-i18next та можливість завантаження перекладів з локальних JSON-файлів або через сервіс Locize.

У межах проєкту i18next використовується для:

- ініціалізації двох мов інтерфейсу - англійської (en) та української (uk) - у файлі i18n.js;

- автоматичного визначення мови браузера через плагін i18next-browser-languagedetector;

- збереження вибраної мови у localStorage для відновлення між сесіями;

- розділення перекладів за просторами назв common (загальні рядки) та monitoring (специфічні до домену) у каталогах en, uk;

- підключення хмарного сервісу Locize через модуль завантаження перекладів i18next-locize-backend для управління перекладами без зміни коду.

3.9 Бібліотека візуалізації числових даних

Для побудови графіків у клієнтській частині могли бути використані Chart.js, ECharts або D3.js, однак для NeverDown обрано Recharts. Бібліотека добре інтегрується з React, оскільки графіки описуються декларативно та складаються з окремих компонентів, як і решта інтерфейсу.

У системі моніторингу графіки потрібні для аналізу зміни стану сервісу в часі. На сторінці деталізації монітора Recharts використовується для відображення часу відповіді, динаміки перевірок і статистичних показників. Це дозволяє користувачу бачити не лише поточний статус, а й періоди деградації або простою.

Recharts побудована поверх D3.js і підтримує лінійні, стовпчасті та інші типи графіків. Компонент ResponsiveContainer забезпечує адаптивне масштабування, тому графіки коректно відображаються в різних розмірах контейнера [9].

У межах проєкту Recharts використовується для:

- побудови графіка часу відповіді (Response Time Chart) на сторінці деталей монітора;

- візуалізації відсотка доступності на пов’язаних блоках статистики;

- декларативної композиції графіків із готових елементів бібліотеки Recharts;

- користувацьких підказок, що показують детальну інформацію про конкретну точку даних.

3.10 Протокол двостороннього обміну даними у реальному часі

Система моніторингу повинна оперативно повідомляти користувача про падіння та відновлення сервісів, оскільки затримка в отриманні інформації збільшує час реагування на інцидент і може ускладнити його усунення. Якщо клієнтська частина буде лише періодично опитувати сервер, повідомлення про проблему може надходити із запізненням. Водночас надто часті запити створюватимуть додаткове навантаження на серверну частину та мережу.

Для розв'язання цієї задачі використовується канал обміну даними у реальному часі. Він дозволяє серверу самостійно надсилати події клієнту одразу після зміни стану монітора, без очікування наступного запиту з браузера. Наприклад, якщо перевірка ресурсу завершилася помилкою або сервіс знову став доступним, серверна частина може миттєво передати відповідне повідомлення користувачу. Після отримання такої події клієнтська частина оновлює статус монітора та відображає сповіщення в інтерфейсі.

У NeverDown для цього використовується SignalR. Ця технологія забезпечує двосторонній обмін даними між сервером і клієнтом, використовуючи WebSocket або інші доступні транспорти. SignalR приховує більшість технічних деталей встановлення та підтримки з'єднання, а також підтримує автоматичне перепідключення, що важливо при нестабільній мережі або короткочасній втраті зв'язку. Крім того, він добре інтегрується з .NET-серверною частиною, тому є зручним вибором для реалізації сповіщень і своєчасного оновлення даних у системі NeverDown.

Таблиця 3.5 – Порівняння протоколів і засобів передачі подій у реальному часі

Критерій	**SignalR**	**WebSocket**	**Server-Sent Events**
Напрям передачі	Двосторонній обмін	Двосторонній обмін	Переважно сервер → клієнт

Кінець таблиці 3.5

Fallback-механізми	Вбудований перехід на SSE або long polling	Відсутні, потрібно реалізовувати окремо	Не потрібні для базового сценарію, але можливості обмежені
Автоматичне перепідключення	Підтримується в клієнтській бібліотеці	Потрібно реалізовувати вручну	Потрібно контролювати вручну
Інтеграція з .NET	Нативна інтеграція з хабами ASP.NET Core	Потребує власної реалізації протоколу повідомлень	Підходить лише для простіших односторонніх подій
Доцільність для NeverDown	Оптимальний вибір для сповіщень і статусів моніторів	Гнучкий, але потребує більше інфраструктурного коду	Недостатній для повноцінної двосторонньої взаємодії

SignalR обрано як компроміс між продуктивністю WebSocket і надійністю готової транспортної абстракції. Для NeverDown це дає змогу швидко доставляти сповіщення користувачу та зберігати працездатність у мережевих умовах, де пряме WebSocket-з’єднання може бути недоступним.

SignalR - обгортка над WebSocket від Microsoft, інтегрована у платформу .NET [10]. Особливості: автоматичний резервний перехід до Server-Sent Events або довге опитування, якщо WebSocket недоступний; вбудована підтримка автоматичного перепідключення; передача access-токена через `accessTokenFactory`.

У межах проєкту SignalR використовується для:

- встановлення WebSocket-з’єднання з серверним хабом notifications у файлі notificationHub.ts;

- передачі JWT-access-токена при кожному підключенні;

- автоматичного перепідключення з експоненціальним зменшенням частоти ;

- прийому подій ReceiveAlert (зміна статусу монітора) та ReceiveNotification (загальні текстові сповіщення);

- подвійної де-дуплікації подій: на рівні з’єднання через Map<callback, wrapper> та на рівні UI через 5-секундне часове вікно у компоненті NotificationHandler.

Канал реального часу доповнює звичайні HTTP-запити, але не замінює їх повністю. Початкове завантаження сторінки все одно виконується через API, оскільки користувачу потрібно отримати актуальний стан моніторів, історію перевірок або перелік інцидентів. SignalR використовується для подій, які виникають після завантаження сторінки і мають бути доставлені без затримки.

Важливо, що інтерфейс повинен коректно поводитися під час розриву з’єднання. Автоматичне перепідключення дозволяє не вимагати від користувача ручного оновлення сторінки, а локальне сховище останніх сповіщень зменшує ризик втрати контексту. Якщо подія надходить повторно, механізм де-дуплікації не дозволяє створити декілька однакових повідомлень у центрі сповіщень.

3.11 Фонове обчислення статистики моніторингу

Альтернативою Web Worker є виконання агрегацій безпосередньо у головному потоці або перенесення всіх розрахунків на серверну частину. Перший варіант може блокувати інтерфейс під час обробки великої історії перевірок, а другий збільшує навантаження на сервер і кількість API-запитів. Web Worker дає змогу залишити інтерфейс плавним, виконуючи статистичні обчислення паралельно з роботою основного потоку браузера.

Web Worker - стандартна технологія браузерів, що дозволяє виконувати JavaScript у фоновому потоці, не блокуючи головний UI-потік [11]. Це важливо для важких обчислень, які можуть займати десятки чи сотні мілісекунд.

У межах проєкту Web Worker використовується для:

- обчислення відсотка доступності, середнього часу відповіді, перцентилів p95/p99 у файлі monitorStats.worker.js;

- агрегації даних історії перевірок у часові «бакети» для відображення на графіку;

- аналізу періодів простою (downtimes) шляхом сканування послідовності перевірок;

- обчислення тренду показників (зростання/спад/стабільність) через порівняння середніх значень двох половин даних;

- забезпечення плавного інтерфейсу під час обробки тисяч точок історії через хук useMonitorStatsWorker.ts із паттерном request-response та 10-секундним timeout.

Обчислення статистики на клієнті може стати помітним навантаженням, коли історія перевірок містить багато точок. Якщо виконувати агрегацію у головному потоці, користувач може відчути затримку під час прокручування сторінки або зміни фільтра часового діапазону. Перенесення таких розрахунків у Web Worker дозволяє залишити інтерфейс доступним для взаємодії.

Практична перевага такого підходу полягає у відокремленні обчислювальної логіки від React-компонентів. Компонент сторінки деталізації передає у worker вхідні дані та отримує готові агреговані показники, не знаючи деталей розрахунку. Це спрощує тестування і дає змогу змінювати алгоритм статистики без зміни візуального шару.

3.12 Засоби стилізації клієнтського інтерфейсу

Для стилізації інтерфейсу системи NeverDown обрано комбінований підхід, що поєднує SCSS Modules, Tailwind CSS і Bulma. Це дозволяє використовувати ізольовані стилі для складних компонентів, швидко налаштовувати типові елементи інтерфейсу та застосовувати готову базову стилізацію.

SCSS Modules використовуються для компонентів, де потрібна чітка структура стилів і відсутність конфліктів між класами. Tailwind CSS застосовується для швидкого налаштування відступів, вирівнювання, розмірів, кольорів і flex/grid-розмітки. Bulma використовується як базовий CSS-фреймворк для стандартних елементів, зокрема форм, таблиць, кнопок і карток.

Отже, комбінований підхід до стилізації дає змогу поєднати швидкість розробки, повторне використання готових стилів і контроль над зовнішнім виглядом окремих компонентів. Це доцільно для NeverDown, оскільки інтерфейс містить як прості UI-елементи, так і складні екрани з таблицями, графіками, формами та статусами моніторів.

У межах проєкту Tailwind застосовується для швидких атомарних правок розмітки, таких як відступи, вирівнювання та типографія [12]. SCSS Modules використовуються для основних стилів компонентів, де потрібна складна логіка вкладеності та підтримка SCSS-змінних [13]. Bulma підключено на рівні index.tsx як стилевий фон для типових елементів форм і таблиць [14].

Іконки інтерфейсу надаються бібліотекою lucide-react у поєднанні з FontAwesome для окремих нестандартних випадків.

Комбінування декількох засобів стилізації потребує дисципліни в організації класів і компонентів. Tailwind доцільно застосовувати для швидкого компонування та простих візуальних станів, тоді як SCSS Modules краще підходять для компонентів із вкладеною структурою, адаптивними станами або складнішою логікою наведення. Bulma при цьому виконує роль базового шару для типових елементів, щоб не створювати з нуля стандартні форми і таблиці.

Для користувача результатом такого підходу є узгоджений інтерфейс: однакові кнопки дій, передбачувані відступи, зрозумілі статуси та стабільне розташування елементів на різних сторінках. У системі моніторингу це важливо, оскільки користувач часто порівнює багато однотипних об’єктів і має швидко знаходити критичну інформацію.

4 ДЕМОНСТРАЦІЯ РОБОТИ СИСТЕМИ

У цьому розділі наведено демонстрацію роботи клієнтської частини системи NeverDown. Описано основні сторінки інтерфейсу та послідовність дій користувача: перегляд головної сторінки, реєстрація, вхід у систему, редагування профілю, створення моніторів, перегляд деталізації, робота з інцидентами, налаштування сповіщень і перемикання мови інтерфейсу.

Кожен підрозділ містить опис окремої частини інтерфейсу, її призначення та основні функціональні можливості.

4.1 Головна сторінка та автентифікація користувача

Перше, на що потрапляє користувач - це головна сторінка сайту. Інтерфейс інтуїтивно зрозумілий, тут користувач отримує перше уявлення про функціональність системи. Основна мета сторінки - надати доступ до базових дій: реєстрація, вхід, ознайомлення з описом платформи та її ключовими можливостями.

На рисунку 4.1 наведено головну сторінку системи з основним інформаційним блоком та статистикою платформи.

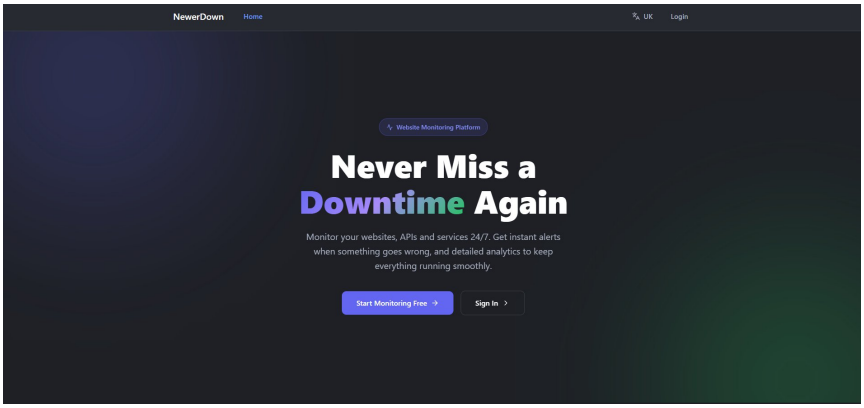

Рисунок 4.1 – Головна сторінка сайту

На головній сторінці відображено hero-блок із закликом «Never Miss a Downtime Again», статистикою платформи, огляд шести ключових можливостей (Uptime Monitoring, Smart Alerts, Detailed Analytics, Global Checks, Incident Tracking, Blazing Fast), live-preview демо-таблицею моніторів, поясненням процесу «Up and Running in Minutes» та повторним CTA-блоком. Лендінг побудований із використанням анімованих лічильників та CSS-переходів для покращення сприйняття користувачем.

Далі користувач може створити новий акаунт або увійти в наявний. Для створення акаунту необхідно ввести ім’я користувача, електронну пошту та двічі підтвердити пароль.

На рисунку 4.2 наведено сторінку реєстрації користувача з полями для створення нового облікового запису.

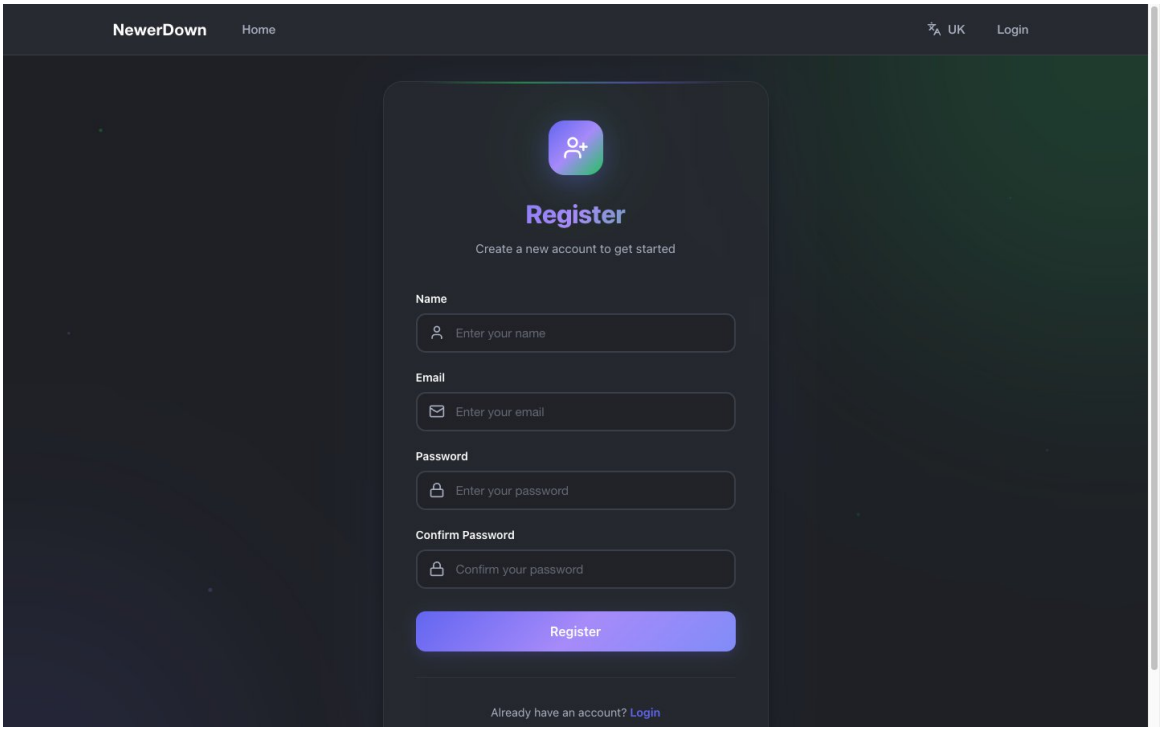

Рисунок 4.2 – Сторінка реєстрації користувача

Форма реєстрації побудована за допомогою бібліотеки `react-hook-form` із декларативною валідацією: ім’я користувача має мати мінімум 2 символи, email перевіряється регулярним виразом, пароль повинен містити щонайменше 8 символів, а Confirm Password - співпадати з основним полем Password. Усі тексти повідомлень про помилки локалізовані через i18next.

На рисунку 4.3 наведено приклад перевірки коректності підтвердження пароля під час реєстрації.

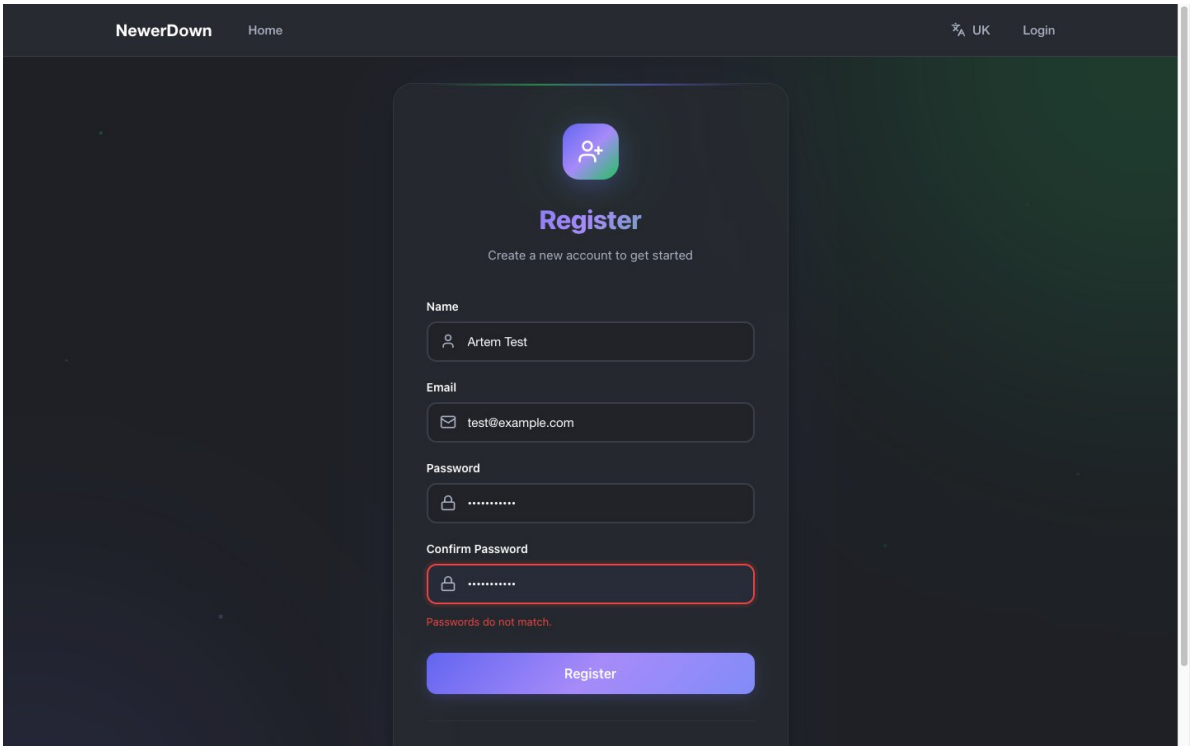

Рисунок 4.3 – Валідація поля Confirm Password при невідповідності паролів

При натисканні на «Register» з різними значеннями полів Password та Confirm Password під полем підтвердження з’являється повідомлення «Passwords do not match.», а саме поле виділяється червоною рамкою. Це класичний приклад декларативної залежної валідації без додаткового коду.

Для входу користувачеві потрібно ввести лише пошту та пароль. У формі реалізовано перемикач показу пароля (іконка ока), що дозволяє за потреби побачити введене значення.

На рисунку 4.4 наведено сторінку входу користувача до системи.

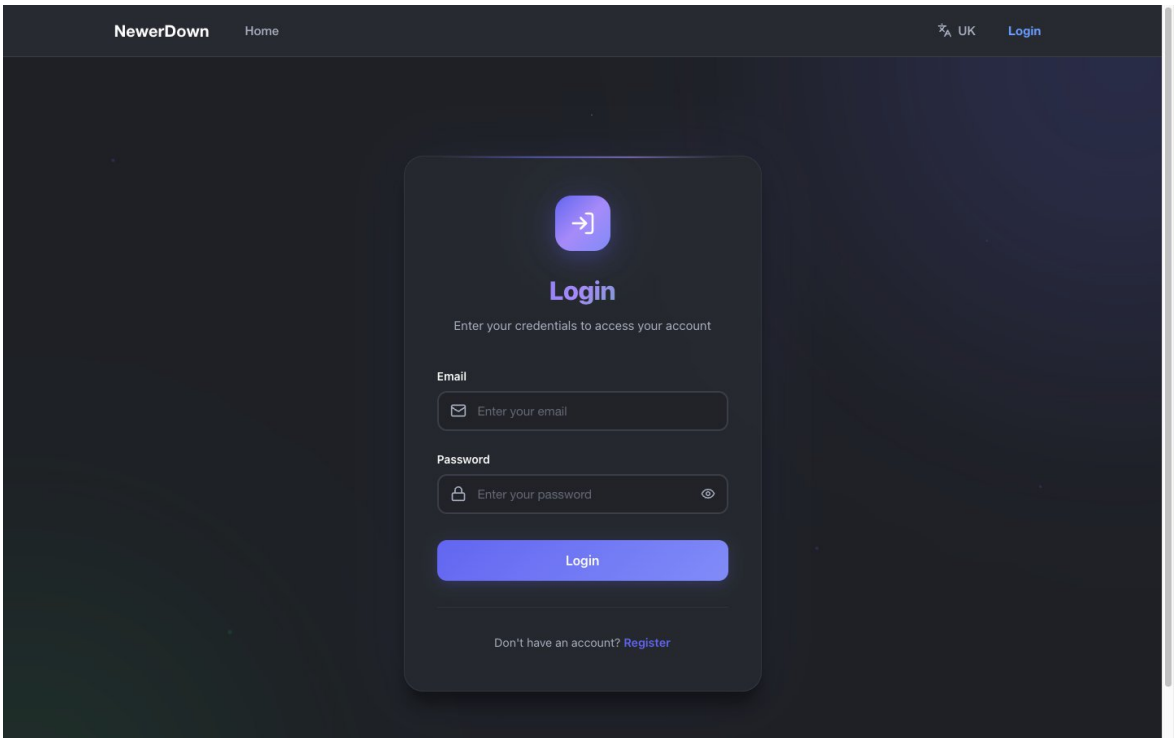

Рисунок 4.4 – Сторінка входу

Сторінка входу містить простий мінімалістичний дизайн із двома полями та основною кнопкою «Login». Після успішної автентифікації клієнт отримує JWT access-токен (зберігається у `localStorage` під ключем `token`) та refresh-токен (зберігається браузером у HttpOnly cookie, що автоматично передається при наступних запитах через прапор `withCredentials: true` у axios-клієнті). Після успішного входу користувача перенаправляють на сторінку профілю або на маршрут, з якого його було перенаправлено компонентом `<RequireAuth>`.

4.2 Сторінка профілю користувача

Після авторизації відкривається сторінка профілю, на якій користувач бачить власні дані (ім’я користувача, пошту), може відредагувати їх, додати додаткову

інформацію (мову, організацію, телефон) та змінити пароль. Верхня панель відображає повну навігацію (Home, Monitoring, Incidents, Alerts), перемикач мови, дзвіночок сповіщень, посилання на акаунт та кнопку Logout.

На рисунку 4.5 наведено сторінку профілю користувача з формою редагування персональних даних.

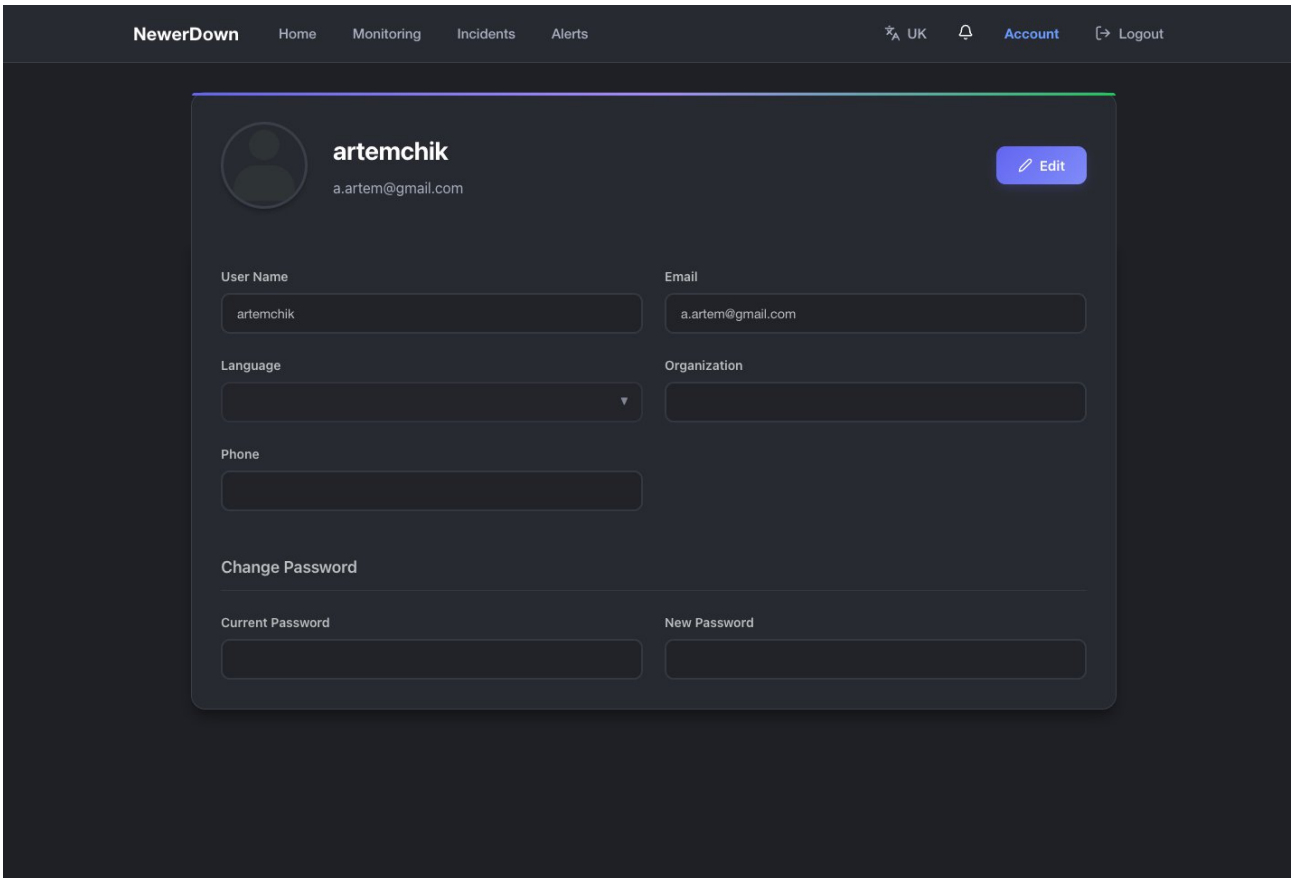

Рисунок 4.5 – Сторінка профілю користувача

Форма редагування профілю автоматично заповнюється даними поточного користувача, що дозволяє швидко переглянути та за потреби оновити персональну інформацію. У ній відображаються основні поля облікового запису, зокрема ім’я користувача, електронна пошта, організація, мова інтерфейсу та номер телефону.

Поле Phone валідується за допомогою регулярного виразу для українських номерів. Воно підтримує кілька форматів введення: із пробілами, дефісами або без роздільників, що робить форму зручнішою для користувача. Окремий блок Change

Password дає змогу безпечно змінити пароль із перевіркою поточного пароля та валідацією нового значення. У правому верхньому куті блоку профілю розташована кнопка Edit, яка переводить форму в режим редагування та відкриває можливість змінювати дані користувача.

4.3 Інтерфейс керування моніторами

Перейшовши у пункт «Monitoring», користувач потрапляє на список своїх моніторів. На прикладі створено два монітори - «KPI website» (для офіційного сайту НТУУ КПІ) та «GitHub HTTPS» - обидва у статусі Active.

На рисунку 4.6 наведено список створених моніторів із поточними статусами та основними діями керування.

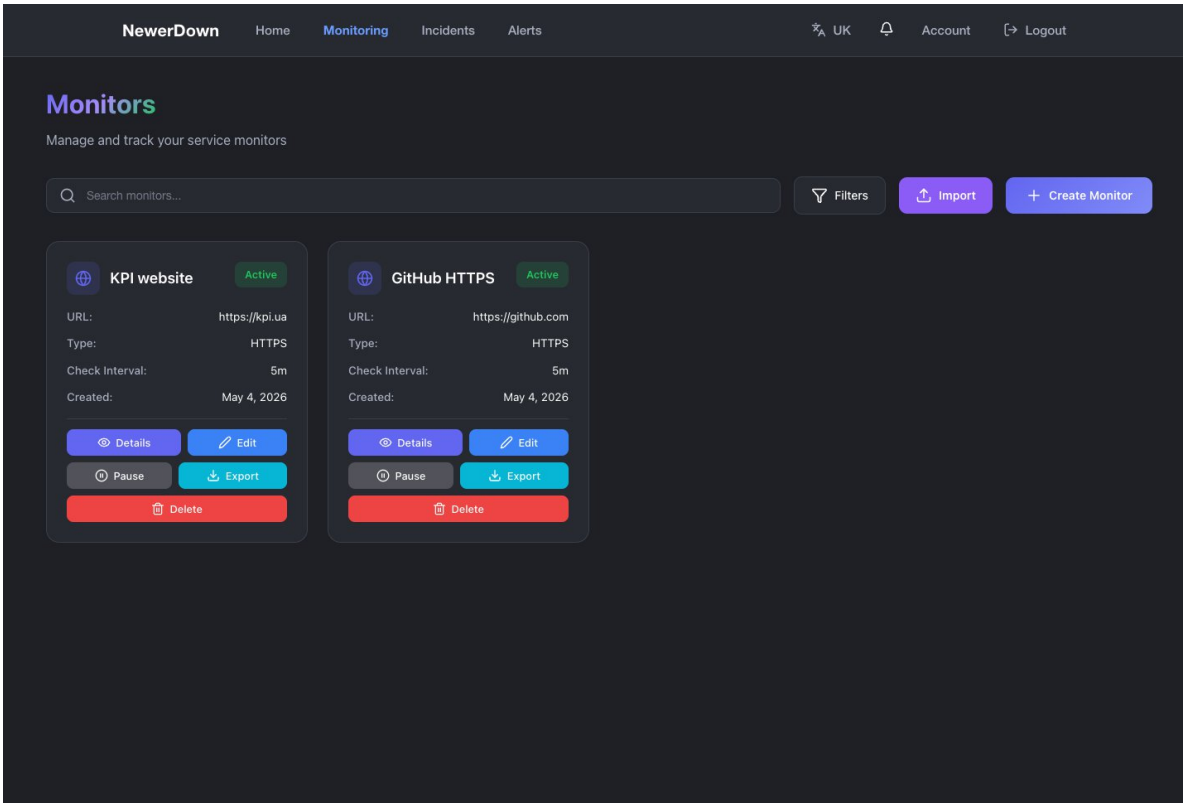

Рисунок 4.6 – Список моніторів зі створеними записами

Угорі сторінки розташовано рядок пошуку, кнопки фільтрів, імпорту з CSV-файлу та створення нового монітора. Кожна картка монітора відображає назву, статус (Active/Paused), URL, тип, інтервал перевірки та дату створення. Внизу картки розташовано кнопки дій: Details - перехід на сторінку деталізації; Edit - редагування параметрів; Pause/Resume - призупинення або відновлення моніторингу; Export - вивантаження конфігурації монітора у CSV-файл; Delete - видалення монітора з підтвердженням. Список підтримує пошук та фільтрацію за статусом, типом перевірки та інтервалом, реалізовану на стороні клієнта через React-хук `useMemo`.

Щоб створити новий монітор, користувач натискає «Create Monitor» і потрапляє на форму, де треба вказати назву, цільову URL/IP-адресу, тип перевірки (HTTP/HTTPS/TCP/PING), порт, інтервал перевірок та опційний прапор «Start monitoring immediately».

На рисунку 4.7 наведено форму створення нового монітора з параметрами перевірки доступності.

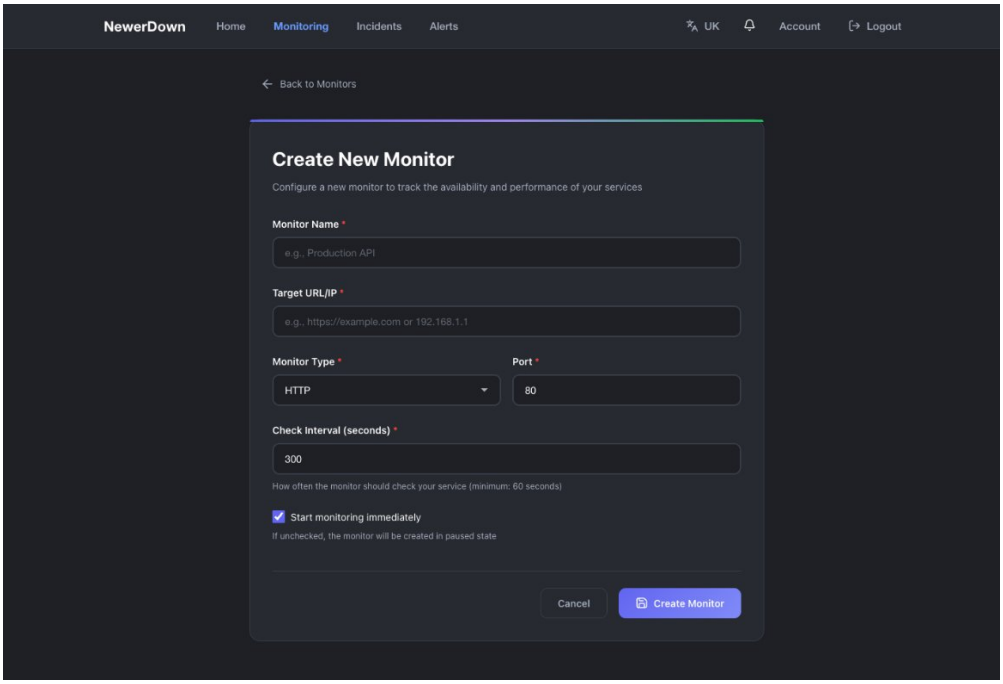

Рисунок 4.7 – Форма створення нового монітора

Форма використовує бібліотеку `react-hook-form` із валідацією: ім’я монітора має містити від 3 до 100 символів; URL перевіряється регулярним виразом, що приймає `https://example.com`, IP-адреси `192.168.1.1` та доменні імена з опційним портом і шляхом; порт повинен бути в діапазоні 1–65535; інтервал - від 60 секунд до 86400 (24 години). За замовчуванням обрано тип HTTP, порт 80 та інтервал 5 хвилин.

Користувач заповнює поля своїм цільовим сервісом - у цьому прикладі моніторимо доступність сайту GitHub.com через HTTPS на стандартному порту 443.

На рисунку 4.8 наведено приклад заповнення форми створення монітора для перевірки HTTPS-ресурсу.

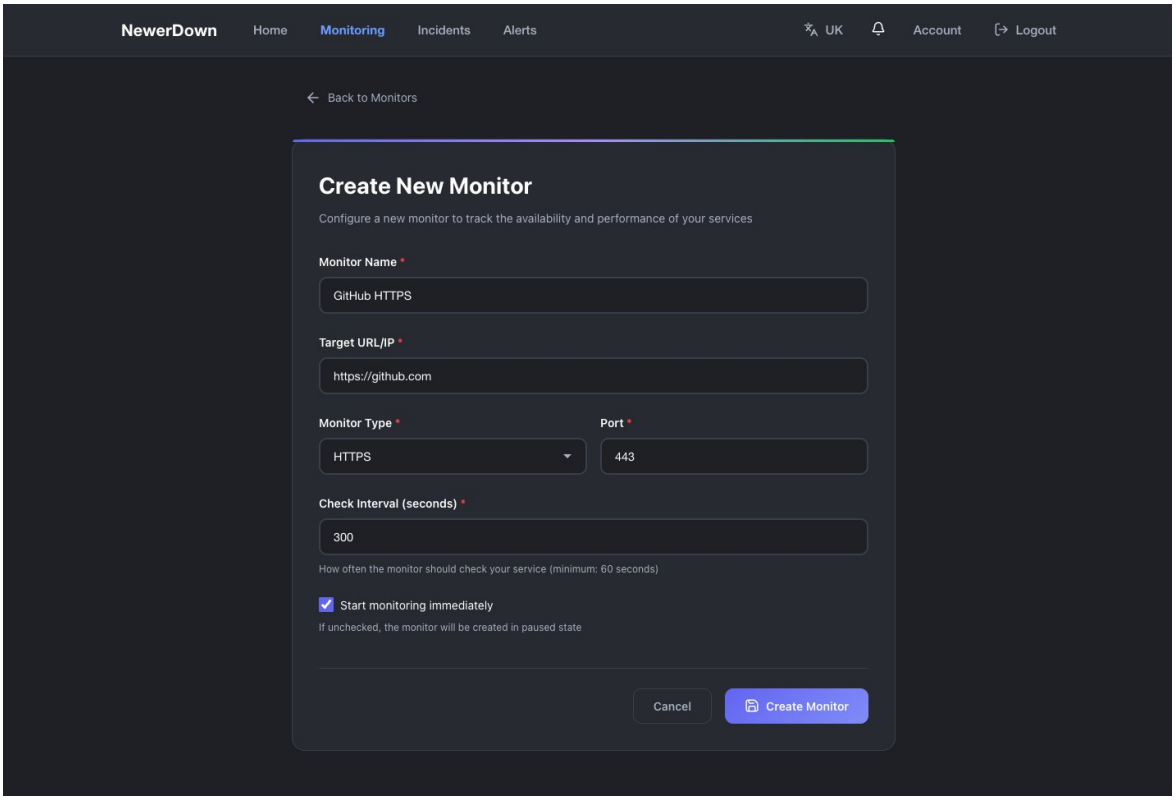

Рисунок 4.8 – Заповнена форма створення монітора

Після натискання кнопки «Create Monitor» виконується Redux-дія `dispatch(createMonitor(data))`, яка через axios-клієнт надсилає POST-запит до

точки доступу API `/api/monitors`. У разі успішної відповіді `extraReducers` слайсу `monitorsSlice` додає новий монітор у початок масиву, а `useNavigate('/monitoring')` повертає користувача на список.

4.4 Сторінка деталізації монітора

Натиснувши «Details» на картці монітора, користувач потрапляє на сторінку деталізації, де відображено поточний статус, узагальнюючу статистику, інтерактивний графік часу відповіді, історію всіх перевірок та перелік періодів простою.

На рисунку 4.9 наведено сторінку деталізації монітора зі статистичними показниками, графіком часу відповіді та історією перевірок.

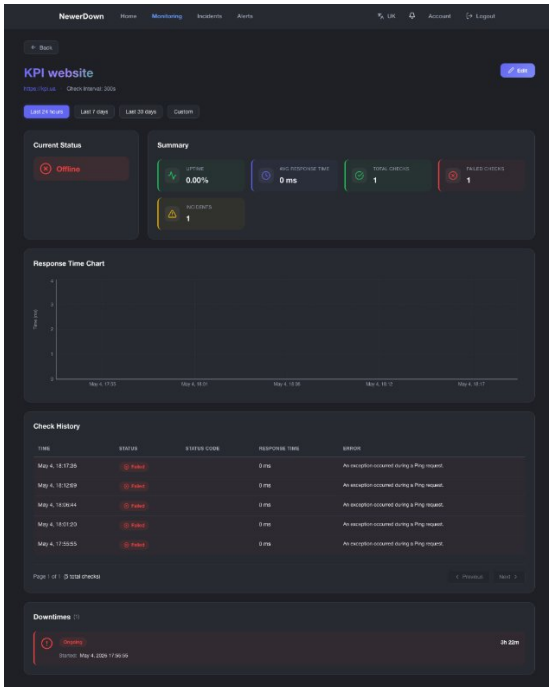

Рисунок 4.9 – Сторінка деталізації монітора зі статистикою та історією перевірок

На сторінці видно поточний статус (Offline), фільтр часового діапазону (Last 24 hours, Last 7 days, Last 30 days, Custom), блок Summary з ключовими метриками (Uptime, Avg Response Time, Total Checks, Failed Checks, Incidents), Response Time Chart на основі бібліотеки Recharts, таблицю Check History з пагінацією та блок Downtimes з активним періодом простою. Дані завантажуються послідовно, щоб уникнути одночасного перевантаження сервера: спочатку статус, потім узагальнення, далі історія, нарешті – простої та графік. Кожен блок має власний прапор завантаження, що дозволяє показувати спінер у конкретному блоці, тоді як інші вже відображають дані.

Сторінка деталізації є одним із найважливіших екранів системи, оскільки саме на ній користувач переходить від загального статусу до аналізу причини проблеми. Показники доступності, середнього часу відповіді та кількості невдалих перевірок дають коротке уявлення про стан сервісу, а графік допомагає побачити, чи була проблема одиничною, повторюваною або поступово наростала.

Фільтр часового діапазону дозволяє аналізувати дані з різною деталізацією. Останні 24 години корисні для оперативного реагування, тижневий або місячний інтервал дає змогу оцінити стабільність сервісу, а власний діапазон потрібний для перевірки конкретного періоду інциденту. Такий набір режимів робить сторінку придатною як для швидкого перегляду, так і для детальнішого аналізу.

4.5 Інтерфейс роботи з інцидентами

При виявленні серверною частиною систематичних помилок перевірок створюється інцидент. Перейшовши до пункту «Incidents», користувач бачить перелік відкритих та розв’язаних інцидентів зі своїх моніторів.

На рисунку 4.10 наведено список інцидентів, пов’язаних із недоступністю відстежуваних ресурсів.

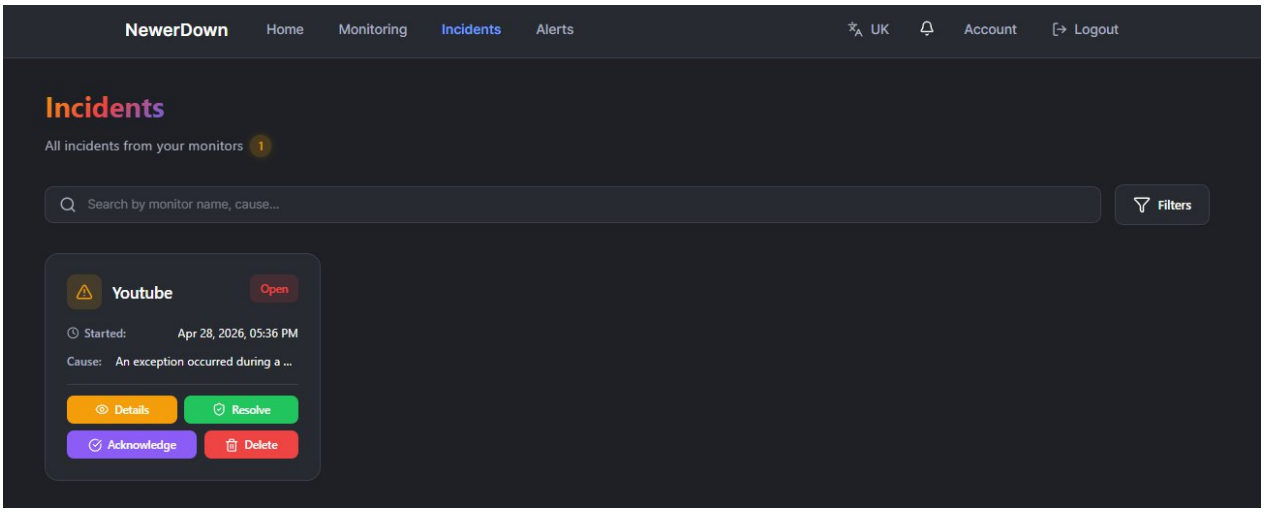

Рисунок 4.10 – Список інцидентів

На картці інциденту відображається основна інформація, необхідна для швидкої оцінки проблеми: час початку, причина виникнення (Cause) та поточний стан інциденту. Також користувачу доступні основні дії: Details для переходу на сторінку деталізації, Resolve для розв’язання інциденту, Acknowledge для підтвердження того, що відповідальна особа вже ознайомлена з проблемою, та Delete для видалення запису.

У верхній частині сторінки розташовано рядок пошуку, фільтри за статусом і елементи сортування. Це спрощує роботу з великою кількістю інцидентів і дозволяє швидко знайти потрібний запис. Інциденти отримуються від серверної частини через GraphQL-точку доступу API /graphql шляхом виконання запиту incidents. Такий підхід дозволяє отримувати саме ті поля, які потрібні для відображення списку.

Після натискання на кнопку Details користувач переходить на сторінку деталізації інциденту. На ній відображається повна інформація про проблему, її причина, статус, пов’язані часові дані, а також форма для додавання коментарів. Це дає змогу не лише переглянути факт збою, а й зафіксувати хід його обробки.

На рисунку 4.11 наведено сторінку деталізації інциденту з інформацією про причину, стан та коментарі.

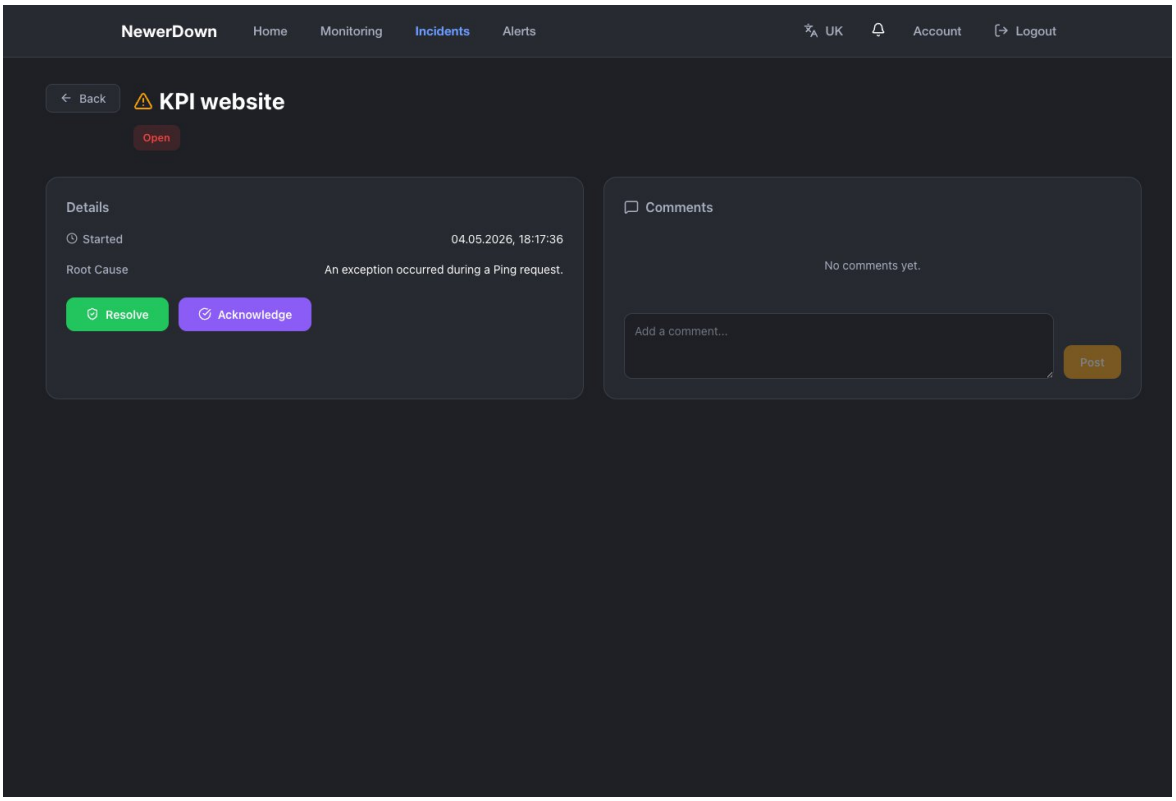

Рисунок 4.11 – Сторінка деталізації інциденту

На сторінці видно блок Details (час початку, Root Cause «An exception occurred during a Ping request.»), кнопки Resolve і Acknowledge, а також блок Comments із полем «Add a comment…». Усі дії з інцидентами реалізовані через GraphQL-мутації: `acknowledgeIncident` для підтвердження, `resolveIncident` для розв’язання з фіксацією першопричини та коментаря, `commentIncident` для додавання довільних коментарів, `deleteIncident` для видалення.

Робота з інцидентами має бути не лише технічною, а й процесною. Підтвердження інциденту показує, що відповідальна особа вже взяла його до уваги, а розв’язання з коментарем фіксує підсумок обробки проблеми. Завдяки цьому історія інциденту стає зрозумілою для інших учасників команди і може використовуватися під час подальшого аналізу.

Коментарі до інциденту виконують роль короткого журналу дій. У них можна зазначити, які перевірки були виконані, чи підтвердилася недоступність сервісу, які зміни внесено на стороні інфраструктури або застосунку. Навіть у навчальному

прототипі така функція демонструє наближення системи до реальних процесів експлуатації програмного забезпечення.

4.6 Налаштування правил сповіщень

Окрім сповіщень у реальному часі у браузері (SignalR), користувач може налаштувати додаткові канали сповіщень через сторінку Alerts. Перехід на цю сторінку показує перелік усіх налаштованих правил сповіщень.

На рисунку 4.12 наведено сторінку керування правилами сповіщень користувача.

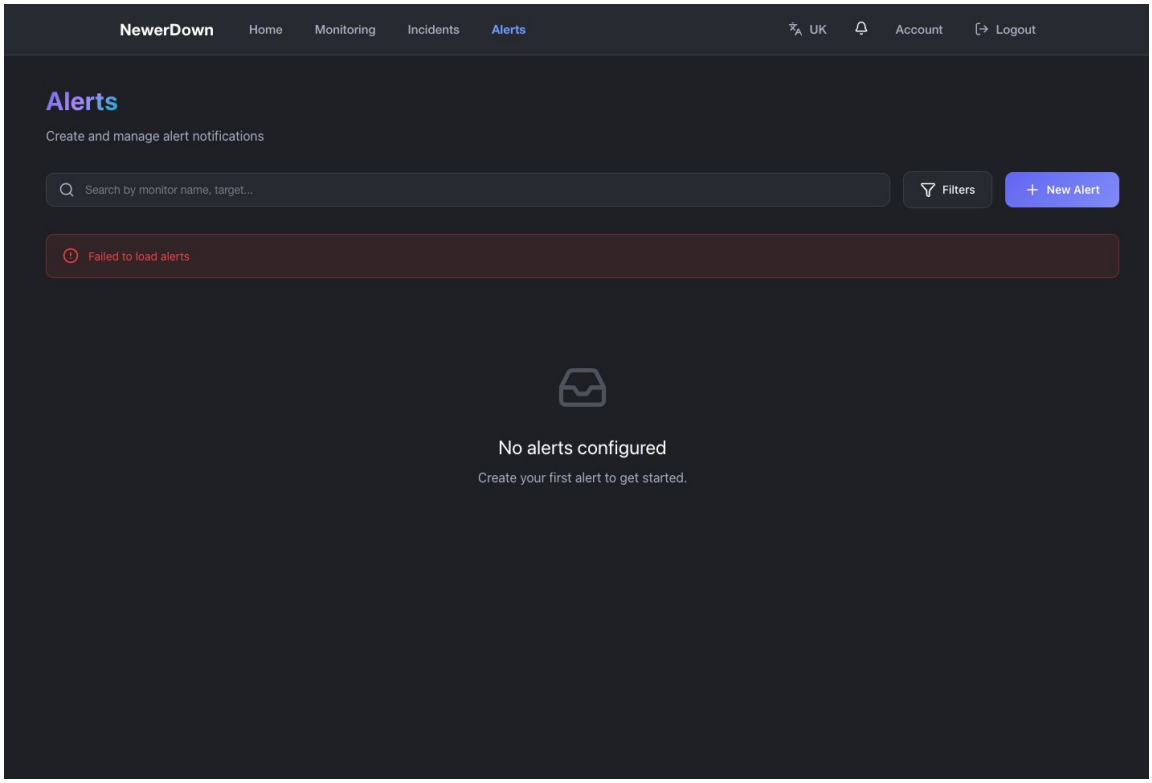

Рисунок 4.12 – Сторінка управління алертами

Сторінка містить рядок пошуку, фільтри та кнопку New Alert. У свіжому акаунті список порожній - система пропонує створити перше правило. Кнопка «New Alert» відкриває модальне вікно для конфігурування правила.

На рисунку 4.13 наведено модальне вікно створення нового правила сповіщення.

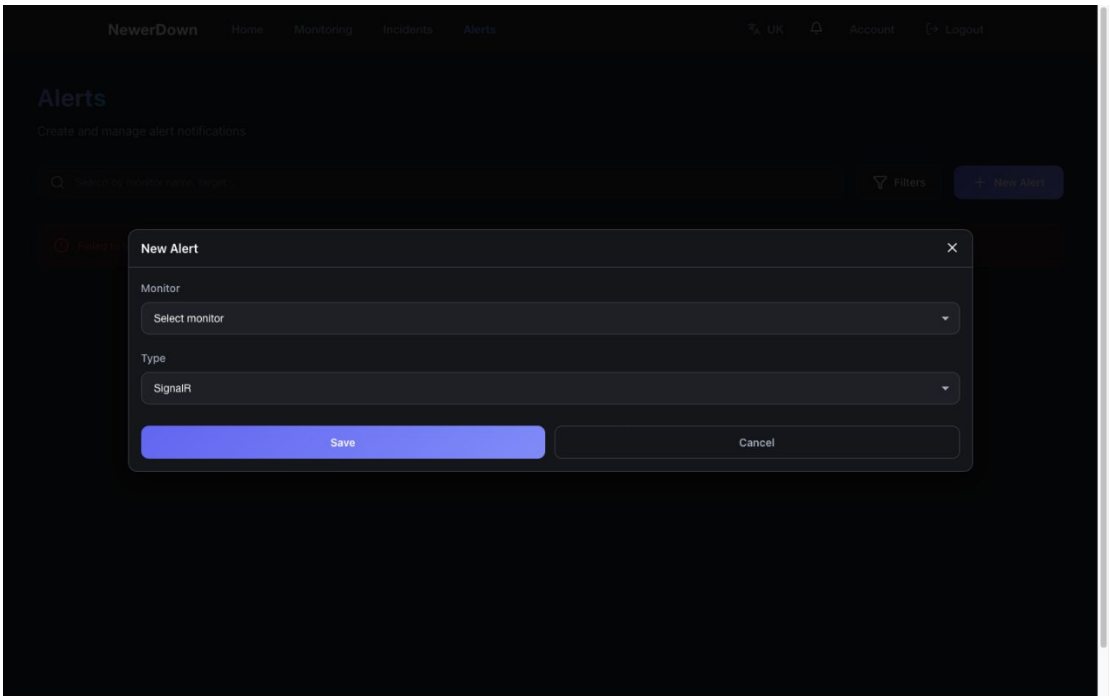

Рисунок 4.13 – Модальне вікно створення нового алерту

У модальному вікні користувач обирає монітор (зі списку всіх своїх моніторів) та тип каналу сповіщення: SignalR (у реальному часі в браузері), Email або Webhook. Для типів Email та Webhook з’являється додаткове поле Target, де вказується відповідно email-адреса або URL вебхука.

На рисунку 4.14 наведено приклад налаштування правила сповіщення електронною поштою.

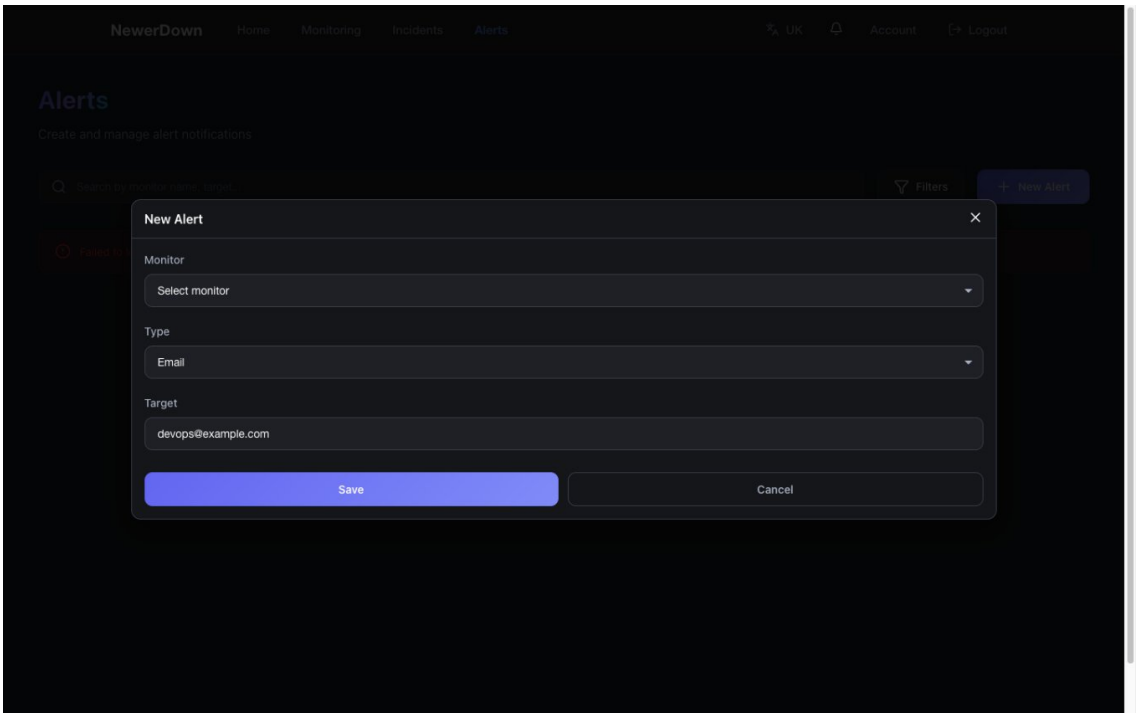

Рисунок 4.14 – Модальне вікно з обраним типом Email та заповненим Target

Після обрання Email у Target вводиться адреса пошти отримувача (у прикладі - `devops@example.com`). Після натискання «Save» виконується POST-запит до точки доступу API `/api/alerts`, а нове правило з’являється у списку. У подальшому, при кожному падінні відповідного монітора, серверна частина надсилає сповіщення на вказану адресу. Реалізована трирівнева система сповіщень (у реальному часі + email + webhook) дозволяє користувачам налаштовувати канали відповідно до власних робочих процесів.

Наявність декількох каналів сповіщень дозволяє адаптувати систему до різних сценаріїв використання. Повідомлення у браузері зручні під час активної роботи з інтерфейсом, електронна пошта підходить для асинхронного інформування, а Webhook може використовуватися для інтеграції з зовнішніми чатами, сервісами автоматизації або внутрішніми інструментами команди.

Правила сповіщень пов’язані з конкретними моніторами, тому користувач може встановити різну поведінку для різних ресурсів. Наприклад, для критичного API можна налаштувати Email і Webhook, а для допоміжного тестового сервісу

залишити лише повідомлення в інтерфейсі. Такий підхід зменшує інформаційний шум і робить систему більш придатною для щоденного використання.

4.7 Перемикання мови інтерфейсу

Окрім функціональних можливостей, клієнтська частина підтримує дві мови інтерфейсу - англійську та українську. Перемикання здійснюється кнопкою у верхній панелі; вибір зберігається у localStorage та відновлюється при наступних сесіях.

На рисунку 4.15 наведено інтерфейс системи після перемикання на українську мову.

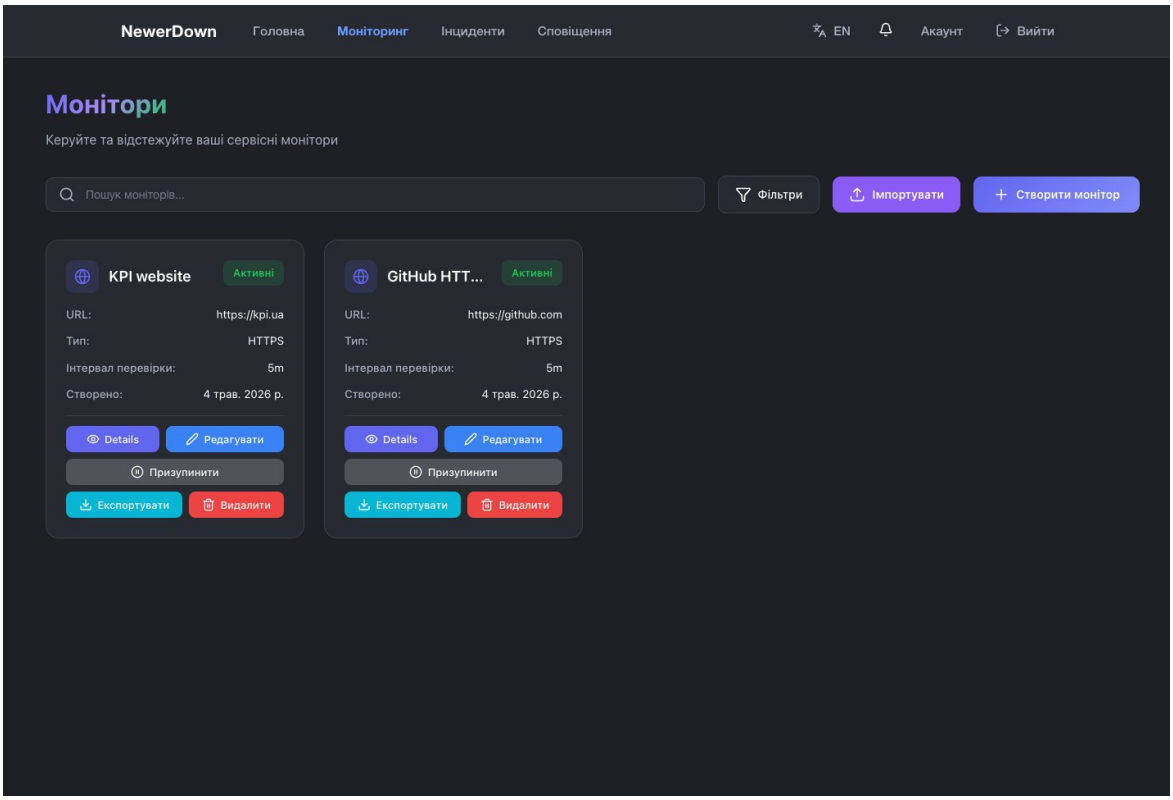

Рисунок 4.15 – Інтерфейс системи українською мовою

При перемиканні на українську мову інтерфейс системи оновлюється одразу, без перезавантаження сторінки. Перекладаються як елементи головної навігації, так і текст усередині робочих сторінок: «Головна», «Моніторинг», «Інциденти», «Сповіщення», «Акаунт», «Вийти», «Монітори», «Керуйте та відстежуйте ваші сервісні монітори», «Пошук моніторів...», «Фільтри», «Імпортувати», «Створити монітор», «Активні», «Тип:», «Інтервал перевірки:», «Створено:», «Редагувати», «Призупинити», «Експортувати», «Видалити» та інші написи.

Переклади організовані у вигляді окремих JSON-файлів, зокрема public/locales/uk/common.json та public/locales/uk/monitoring.json. Такий підхід дозволяє розділити загальні фрази інтерфейсу і тексти, пов’язані безпосередньо з модулем моніторингу. Бібліотека i18next разом із React-хуком useTranslation відстежує зміну поточної мови, тому всі компоненти, які використовують переклади, автоматично оновлюються. Це робить перемикання мови зручним для користувача та не порушує поточний сценарій роботи.

4.8 Центр сповіщень

Окремим важливим компонентом інтерфейсу є центр сповіщень у реальному часі. Він розташований у верхній панелі у вигляді іконки дзвіночка. Якщо користувач має непрочитані повідомлення, біля іконки відображається бейдж із їхньою кількістю. Завдяки цьому користувач одразу бачить, що в системі з’явилися нові події, пов’язані зі станом моніторів або інцидентами.

Після натискання на іконку відкривається випадаюче меню з історією останніх сповіщень. У ньому користувач може швидко переглянути останні події, не переходячи на окрему сторінку. Такий формат зручний для оперативної роботи, оскільки повідомлення про падіння або відновлення сервісу доступні безпосередньо з будь-якого розділу системи. На рисунку 4.16 наведено приклад випадаючого меню центру сповіщень із переліком останніх подій.

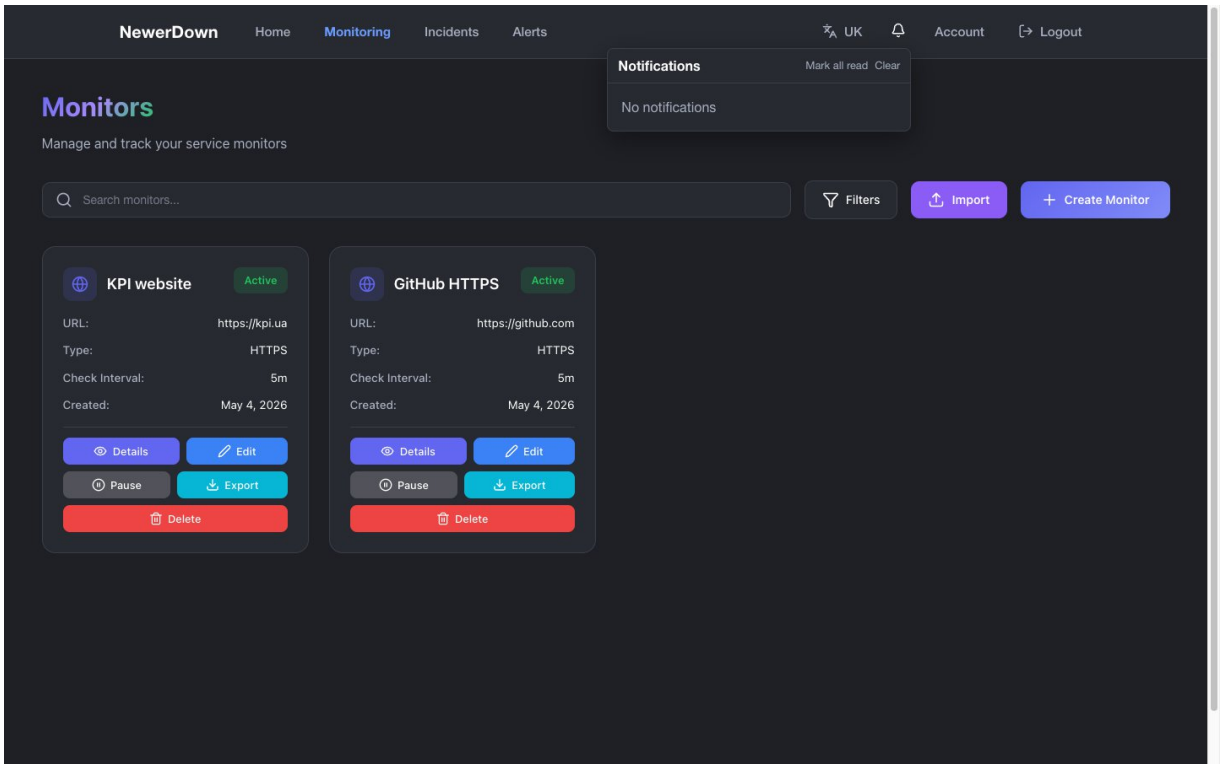

Рисунок 4.16 – Випадаюче меню центру сповіщень

Сповіщення у системі зберігаються в централізованому pub/sub-сховищі notificationStore у файлі src/shared/services/notificationStore.ts. Такий підхід дозволяє різним компонентам застосунку працювати зі сповіщеннями без прямої залежності один від одного. Історія додатково зберігається у localStorage під ключем nd_notifications_history_v1, що дає змогу не втрачати останні повідомлення після оновлення сторінки. Для оптимізації зберігаються лише 20 останніх записів.

Кожне сповіщення містить заголовок, текст, маркер серйозності (info, success або error) та, за потреби, monitorId. Завдяки цьому користувач може натиснути на повідомлення й одразу перейти на сторінку відповідного монітора. У випадаючому меню також передбачено кнопки Mark all read і Clear для позначення повідомлень як прочитаних або повного очищення списку.

Коли SignalR-канал отримує подію ReceiveAlert, компонент NotificationHandler показує toast-повідомлення у нижньому правому куті екрана та додає запис у notificationStore. Для уникнення повторів використовується подвійна

де-дуплікація: на рівні з’єднання та через 5-секундне часове вікно в інтерфейсі. Це запобігає дублюванню сповіщень під час перепідключень або повторного отримання однакових подій.

Центр сповіщень виконує роль короткої історії подій, яка залишається доступною навіть після закриття toast-повідомлення. Це важливо, оскільки користувач може не встигнути прочитати спливне повідомлення або отримати декілька подій поспіль. Переглянувши центр сповіщень, він бачить останні зміни стану моніторів і може перейти до потрібної сторінки.

Збереження обмеженої кількості останніх повідомлень у localStorage є компромісом між зручністю і простотою реалізації. Історія не перетворюється на повноцінний журнал аудиту, але дозволяє не втратити найближчий контекст після оновлення сторінки. Для більш масштабної системи цей механізм можна доповнити серверним зберіганням сповіщень і синхронізацією між пристроями користувача.

Загалом інтерфейс системи забезпечує зручне управління моніторами, інцидентами та каналами сповіщень. Усі основні дії відображаються в інтерфейсі та синхронізуються із серверною частиною, що забезпечує узгоджений стан між клієнтом і сервером.

ВИСНОВКИ

У дипломній роботі виконано поставлені задачі щодо розроблення клієнтської частини вебсистеми моніторингу доступності програмних застосунків NeverDown.

По-перше, було проаналізовано існуючі рішення для моніторингу доступності вебсервісів. Порівняння показало, що готові платформи надають корисні можливості контролю стану ресурсів, перегляду статистики та налаштування сповіщень. Водночас такі рішення можуть бути обмеженими щодо локалізації, гнучкої інтеграції з власною серверною частиною, адаптації під конкретні сценарії роботи та керування інцидентами в межах навчального проєкту. Це підтвердило доцільність створення власної клієнтської частини системи моніторингу.

По-друге, розглянуто основні методи моніторингу: активний синтетичний, пасивний користувацький, агентний та інфраструктурний, а також журнальний і подієвий. Для системи NeverDown обґрунтовано вибір активного синтетичного підходу, оскільки він дозволяє незалежно від реального користувацького трафіку регулярно фіксувати стан ресурсів, накопичувати історію перевірок, визначати періоди недоступності та формувати підстави для створення інцидентів і сповіщень.

По-третє, сформовано вимоги до клієнтської частини. Визначено, що система має забезпечувати автентифікацію користувача, керування моніторами, перегляд поточного стану ресурсів, аналіз статистики доступності, роботу з інцидентами, налаштування правил сповіщень, редагування профілю, підтримку декількох мов інтерфейсу та зручну роботу з результатами моніторингу. Окрему увагу приділено вимогам до зрозумілості інтерфейсу, узгодженості даних між розділами та швидкого доступу до ключової інформації.

По-четверте, спроєктовано архітектуру клієнтської частини. Вона включає шар сторінок, повторно використовувані компоненти, централізований стан, API-

модулі, сервіс подій у реальному часі, фонову обробку статистики, типізовані моделі даних і підсистему перекладу інтерфейсу. Такий підхід забезпечує логічне розділення відповідальностей між частинами застосунку, спрощує підтримку коду та створює основу для подальшого розширення функціональності.

По-п’яте, реалізовано основні сценарії користувача: реєстрацію та вхід, перегляд і створення моніторів, сторінку деталізації з історією та статистикою, роботу з інцидентами, налаштування правил сповіщень, центр повідомлень, профіль користувача та перемикання мови інтерфейсу. Реалізовані можливості охоплюють основний цикл роботи користувача із системою: від додавання ресурсу для моніторингу до перегляду результатів перевірок, аналізу проблем і реагування на інциденти.

Демонстрація роботи системи підтвердила, що розроблена клієнтська частина забезпечує повний цикл взаємодії користувача з результатами моніторингу доступності. Інтерфейс дозволяє отримувати актуальну інформацію про стан ресурсів, переглядати історичні дані, працювати з інцидентами та налаштовувати сповіщення відповідно до потреб користувача.

Отже, мету дипломної роботи досягнуто: створено функціональний клієнтський вебзастосунок NeverDown, який забезпечує зручну роботу з даними моніторингу доступності програмних застосунків і може бути використаний як основа для подальшого розвитку системи. У майбутньому систему можна розширити новими типами перевірок, додатковими каналами сповіщень, розширеною аналітикою та засобами командної роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. React documentation [Електронний ресурс] : офіц. сайт. — Meta Platforms, 2026. — Режим доступу: https://react.dev (дата звернення: 5.04.2026).

2. TypeScript handbook [Електронний ресурс] : офіц. документація. — Microsoft, 2026. — Режим доступу: https://www.typescriptlang.org/docs/ (дата звернення: 11.04.2026).

3. Redux Toolkit official documentation [Електронний ресурс] : офіц. документація. — Redux Team, 2026. — Режим доступу: https://redux-toolkit.js.org/ (дата звернення: 7.04.2026).

4. Vite — Next Generation Frontend Tooling [Електронний ресурс] : офіц. сайт. — Vite Team, 2026. — Режим доступу: https://vite.dev/ (дата звернення: 5.04.2026).

5. Axios documentation [Електронний ресурс] : офіц. документація. — Axios Project, 2026. — Режим доступу: https://axios-http.com/ (дата звернення: 23.04.2026).

6. React Router v7 documentation [Електронний ресурс] : офіц. документація. — Remix Software, 2026. — Режим доступу: https://reactrouter.com/ (дата звернення: 22.04.2026).

7. React Hook Form [Електронний ресурс] : офіц. документація. — React Hook Form Project, 2026. — Режим доступу: https://react-hook-form.com/ (дата звернення: 25.04.2026).

8. i18next internationalization framework [Електронний ресурс] : офіц. документація. — i18next Project, 2026. — Режим доступу: https://www.i18next.com/ (дата звернення: 10.05.2026).

9. Recharts — Composable React charts [Електронний ресурс] : офіц. документація. — Recharts Group, 2026. — Режим доступу: https://recharts.org/ (дата звернення: 11.05.2026).

10. ASP.NET Core SignalR JavaScript client [Електронний ресурс] : документація Microsoft. — Microsoft, 2026. — Режим доступу:

https://learn.microsoft.com/en-us/aspnet/core/signalr/javascript-client (дата звернення: 11.04.2026).

11. Web Workers API [Електронний ресурс] : документація MDN Web Docs. — Mozilla, 2026. — Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API (дата звернення: 25.04.2026).

12. Tailwind CSS documentation [Електронний ресурс] : офіц. документація. — Tailwind Labs, 2026. — Режим доступу: https://tailwindcss.com/ (дата звернення: 15.04.2026).

ДОДАТОК А

«Клієнтська частина вебсистеми моніторингу доступності програмних застосунків»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_23

Реалізація ключових модулів клієнтської частини системи NeverDown
Мова програмування: TypeScript

Аркушів 5

```
import axios, { type AxiosRequestConfig, type Method } from 'axios';
import { refreshToken } from '../api/auth';
import type { AuthResponse } from '../shared/types/Auth';

const BASE_URL = 'https://newerdown.online';

function wait(delay: number) {
  return new Promise((resolve) => setTimeout(resolve, delay));
}

export const instance = axios.create({
  baseURL: BASE_URL,
  headers: {
    'Content-Type': 'application/json; charset=UTF-8',
  },
  withCredentials: true,
});

instance.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token && config.headers) {
    config.headers['Authorization'] = `Bearer ${token}`;
  }
  return config;
});

let isRefreshing = false;
let refreshSubscribers: ((token: string) => void)[] = [];

function subscribeTokenRefresh(cb: (token: string) => void) {
  refreshSubscribers.push(cb);
}

function onRefreshed(newToken: string) {
  refreshSubscribers.forEach((cb) => cb(newToken));
  refreshSubscribers = [];
}

instance.interceptors.response.use(
  (res) => res,
  async (error) => {
    const originalRequest = error.config;

    if (error.response?.status === 401 && error.config && !originalRequest._isRetry) {
      originalRequest._isRetry = true;

      if (isRefreshing) {
        return new Promise((resolve) => {
          subscribeTokenRefresh((newToken) => {
            originalRequest.headers['Authorization'] = `Bearer ${newToken}`;
            resolve(instance(originalRequest));
          });
        });
      }

      isRefreshing = true;

      try {
        console.log('Attempting to refresh token...');
        const response: AuthResponse = await refreshToken();
        console.log('REFRESH RESPONSE:', response);

        const newToken = response.accessToken;
        localStorage.setItem('token', newToken);
        console.log('TOKEN UPDATED IN LOCAL STORAGE:', newToken);

        onRefreshed(newToken);
        isRefreshing = false;

        originalRequest.headers['Authorization'] = `Bearer ${newToken}`;
        return instance(originalRequest);
      } catch (err: unknown) {
        isRefreshing = false;
```

```
      if (axios.isAxiosError(err)) {
        const detail = err.response?.data?.detail;
        console.error('Refresh token error:', detail);

        if (detail === 'Invalid refresh token. Please login again.') {
          localStorage.removeItem('token');
        }
      } else {
        console.error('Unexpected error:', err);
      }

      throw err;
    }
  }

  if (error.config?.url?.includes('/token/refresh') && error.response?.status === 400) {
    console.error('Refresh token invalid. Logging out.');
    localStorage.removeItem('token');
  }

  throw error;
 },
);

async function request<T>(url: string, method: Method = 'GET', data: any = null): Promise<T> {
 await wait(0);

 const config: AxiosRequestConfig = {
  url,
  method,
  data,
 };

 try {
  const response = await instance.request<T>(config);
  return response.data;
 } catch (error: any) {
  if (error.response) {
    console.error('Request error:', error.response.data);
    throw new Error(JSON.stringify(error.response.data));
  } else if (error.request) {
    throw new Error('No response from server');
  } else {
    throw new Error(error.message);
  }
 }
}

async function requestForm<T>(url: string, file: File, fieldName: string = 'file'): Promise<T> {
 await wait(0);

 const formData = new FormData();
 formData.append(fieldName, file);

 try {
  const response = await instance.post<T>(url, formData, {
    headers: {
      'Content-Type': 'multipart/form-data',
    },
  });
  return response.data;
 } catch (error: any) {
  if (error.response) {
    console.error('Request form error:', error.response.data);
    throw new Error(JSON.stringify(error.response.data));
  } else if (error.request) {
    throw new Error('No response from server');
  } else {
    throw new Error(error.message);
  }
 }
}

export const client = {
 get: <T>(url: string) => request<T>(url, 'GET'),
```

```
  post: <T>(url: string, data?: any) => request<T>(url, 'POST', data),
  patch: <T>(url: string, data: any) => request<T>(url, 'PATCH', data),
  put: <T>(url: string, data?: any) => request<T>(url, 'PUT', data),
  delete: (url: string) => request(url, 'DELETE'),

  postForm: <T>(url: string, file: File, fieldName?: string) =>
    requestForm<T>(url, file, fieldName),
};

import { configureStore, type ThunkAction, type Action } from '@reduxjs/toolkit';
import authReducer from '../features/authSlice';
import userAccountReducer from '../features/userAccountSlice';
import monitorsReducer from '../features/monitorsSlice';
import monitorDetailsReducer from '../features/monitorDetailsSlice';

export const store = configureStore({
  reducer: {
    auth: authReducer,
    userAccount: userAccountReducer,
    monitors: monitorsReducer,
    monitorDetails: monitorDetailsReducer,
  },
});

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof store.getState>;

export type AppThunk<ReturnType = void> = ThunkAction<
  ReturnType,
  RootState,
  unknown,
  Action<string>
>;

import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import { login, singUp, changePassword } from '../api/auth';
import type { Login, SingUp, User, ChangePassword } from '../shared/types/Auth';

interface AuthState {
  token: string | null;
  user: User | null;
  loading: boolean;
  error: string | null;
}

const storedToken = localStorage.getItem('token') || null;

const initialState: AuthState = {
  token: storedToken,
  user: null,
  loading: false,
  error: null,
};

export const loginUser = createAsyncThunk(
  'auth/loginUser',
  async (credentials: Login, { rejectWithValue }) => {
    try {
      const response: any = await login(credentials);

      if (
        typeof response === 'string' &&
        response.includes('.') &&
        response.split('.').length === 3
      ) {
        return { token: response };
      } else if (
        typeof response === 'object' &&
        response.token &&
        typeof response.token === 'string' &&
        response.token.includes('.') &&
        response.token.split('.').length === 3
      ) {
        return response;
      } else {
        return rejectWithValue(response || 'Authentication error');
```

```
      }
    } catch (error: any) {
      console.error('Error in loginUser:', error);

      if (error.response?.data?.error?.description) {
        return rejectWithValue(error.response.data.error.description);
      }

      return rejectWithValue(error.message || 'Unknown login error');
    }
  },
);

export const singUpUser = createAsyncThunk(
  'auth/singUpUser',
  async (userData: SingUp, { rejectWithValue }) => {
    try {
      const response = await singUp(userData);
      return response;
    } catch (error: any) {
      console.error('Error in singUpUser:', error);

      if (error.response?.data?.error?.description) {
        return rejectWithValue(error.response.data.error.description);
      }

      return rejectWithValue(error.message || 'Unknown signup error');
    }
  },
);

export const changePasswordUser = createAsyncThunk(
  'auth/changePasswordUser',
  async (passwordData: ChangePassword, { rejectWithValue }) => {
    try {
      const response = await changePassword(passwordData);
      return response;
    } catch (error: any) {
      console.error('Error in changePasswordUser:', error);

      if (error.response?.data?.error?.description) {
        return rejectWithValue(error.response.data.error.description);
      }

      return rejectWithValue(error.message || 'Unknown change password error');
    }
  },
);

const authSlice = createSlice({
  name: 'auth',
  initialState,
  reducers: {
    clearError: (state) => {
      state.error = null;
    },
    logout: (state) => {
      state.token = null;
      state.user = null;
      localStorage.removeItem('token');
    },
  },
  extraReducers: (builder) => {
    builder
      .addCase(loginUser.pending, (state) => {
        state.loading = true;
        state.error = null;
      })
      .addCase(loginUser.fulfilled, (state, action) => {
        state.loading = false;
        state.token = action.payload.token;

        if (action.payload.user) {
          state.user = action.payload.user;
          localStorage.setItem('user', JSON.stringify(action.payload.user));
        }
```

```
        localStorage.setItem('token', action.payload.token);
      })
      .addCase(loginUser.rejected, (state, action) => {
        state.loading = false;
        state.error = action.payload as string;
      })
      .addCase(singUpUser.pending, (state) => {
        state.loading = true;
        state.error = null;
      })
      .addCase(singUpUser.fulfilled, (state, action) => {
        state.loading = false;
        if (action.payload.accessToken) {
          state.token = action.payload.accessToken;
          localStorage.setItem('token', action.payload.accessToken);
          if (action.payload.user) {
            state.user = action.payload.user;
            localStorage.setItem('user', JSON.stringify(action.payload.user));
          }
        }
      })
      .addCase(singUpUser.rejected, (state, action) => {
        state.loading = false;
        state.error = action.payload as string;
      })
      .addCase(changePasswordUser.pending, (state) => {
        state.loading = true;
        state.error = null;
      })
      .addCase(changePasswordUser.fulfilled, (state) => {
        state.loading = false;
        state.token = null;
        state.user = null;
        localStorage.removeItem('token');
      })
      .addCase(changePasswordUser.rejected, (state, action) => {
        state.loading = false;
        state.error = action.payload as string;
      });
  },
});

export const { clearError, logout } = authSlice.actions;
export default authSlice.reducer;
```