Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп'ютерних наук

Спеціальність 125 «Кібербезпека»

Освітня програма «Безпека інформаційних та комунікаційних систем»

«Допущено до захисту»

Зав.кафедрою БІСТ

Сергій РАССОМАХІН

_____

« » 2022 р.

Пояснювальна записка

до кваліфікаційної роботи магістра

на тему: «Дослідження вразливостей безпеки та розробка захищеної системи розумного будинку»

оцінка « »

Голова ЕК

Доценко С.І. _____

Керівник к.т.н. Мелкозьорова О.М.

Рецензент проф., д.т.н. Краснобаєв В.А.

Виконавець : студент групи КБ-61

Воскобойников Д. О.

Харків – 2022

РЕФЕРАТ

Звіт про виконання дипломної роботи: 68 сторінок, 23 рисунки, 1 таблиця, 2 додатка та 30 джерел.

Метою роботи є дослідження вразливостей безпеки та розробка захищеної системи розумного будинку, а також визначення актуальних способів реалізації механізмів захисту системи, задля забезпечення її від найпоширеніших вразливостей веб-застосунків.

У проекті було розглянуто й проаналізовано загальні положення безпеки веб-застосунків, найпоширеніші вразливостей веб-систем, їх види та методи запобігання таким вразливостям. На основі проведеного дослідження та аналізу предметної області було спроектовано веб-застосунок й визначено вимоги до системи. На підставі вимог реалізовано захищену систему розумного будинку на базі клієнт-серверної архітектурної моделі. Безпеку веб-застосунку було перевірено й протестовано різними методами. Тестування показало, що розроблений веб-застосунок дійсно представляє собою систему, що захищена дієвими методами забезпечення безпеки від найпоширеніших вразливостей.

Результати проекту можуть бути використані в якості оглядової роботи сучасного стану методів забезпечення безпеки веб-застосунків, властивих їм загроз та методів захисту від вразливостей, задля подальшої реалізації захищеного веб-застосунку.

Серед можливих напрямків розвитку роботи можна виділити огляд безпеки систем, що реалізовані на інших мовах програмування та використовують нереляційні бази даних, подальше проведення порівняльного аналізу засобів та інструментів реалізації таких систем, задля оцінки їхньої ефективності й захищеності.

Ключові слова: БЕЗПЕКА ВЕБ-ЗАСТОСУНКІВ, ВРАЗЛИВОСТІ ВЕБ-ЗАСТОСУНКІВ, МЕХАНІЗМИ ЗАХИСТУ ВЕБ-ЗАСТОСУНКІВ,

ПРОЕКТУВАННЯ ЗАХИЩЕНОЇ СИСТЕМИ, ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ, REACTJS, NODEJS, MYSQL.

ABSTRACT

Thesis report: 68 pages, 23 figures, 1 table, 2 appendices and 30 sources.

The purpose of the work is the research of security vulnerabilities and the development of a protected smart home system, as well as the determination of current ways of implementing system protection mechanisms to protect it from the most common vulnerabilities of web applications.

The project considered and analyzed the general security provisions of web applications, the most common vulnerabilities of web systems, their types and methods of preventing such vulnerabilities. Based on the conducted research and analysis of the subject area, a web application was designed and system requirements were determined. Based on the requirements, a secure smart home system based on the client-server architectural model was implemented. The security of the web application has been checked and tested using various methods. Testing showed that the developed web application really represents a system protected by effective security methods against the most common vulnerabilities.

The results of the project can be used as a review of the current state of web application security methods, their inherent threats and methods of protection against vulnerabilities, for the further implementation of a secure web application.

Among the possible areas of development of the work, it is possible to single out a review of the security of systems implemented in other programming languages and using non-relational databases, further conducting a comparative analysis of the means and tools for the implementation of such systems, in order to assess their effectiveness and security.

Keywords: WEB APPLICATION SECURITY, WEB APPLICATION VULNERABILITIES, WEB APPLICATION PROTECTION MECHANISMS, SECURE SYSTEM DESIGN, WEB APPLICATION TESTING, REACTJS, NODEJS, MYSQL.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ACL – Access Control List
AJAX – Asynchronous JavaScript And XML
API – Application Programming Interface
ASP – Active Server Pages
CGI – Common Gateway Interface
CI/CD – Continuous Integration/Continuous Delivery
CSS – Cascading Style Sheets
DAST – Dynamic Application Security Testing
DDoS – Distributed Denial of Service
DOM – Document Object Model
FaaS – Function as a Services
FTP – File Transfer Protocol
HTML – HyperText Markup Language
HTTPS – HyperText Transfer Protocol Secure
IAST – Interactive Application Security Testing
JSON – JavaScript Object Notation
JSP – Jakarta Server Pages
JWT – JSON Web Token
MPA – Multiple-Page Application
ORM – Object-Relational Mapping
PaaS – Platform as a Service
PHP – PHP: Hypertext Preprocessor
PWA – Progressive Web Application
RASP – Runtime Application Self Protection
RBAC – Role-Based Access Control
SAST – Static Application Security Testing

SEO – Search Engine Optimization

SPA – Single-Page Application

SSL – Secure Sockets Layer

TLS – Transport Layer Security

UI – User Interface

URL – Uniform Resource Locator

XML – Extensible Markup Language

XSS – Cross-Site Scripting

ОС – Операційна система

ПЗ – Програмне Забезпечення

СКБД – Система Керування Базами Даних

ВСТУП

В сучасну цифрову епоху веб-застосунки стали головним рушієм розвитку усіх сфер мережевих послуг. Зважаючи на те, що веб-застосунків з кожним роком стає все більше й системи стають все більш масштабованими, розробникам ПЗ все важче слідкувати за перебігом змін пов’язаних з забезпеченням безпеки веб-програм та усунення недоліків й вразливостей таких систем. Саме тому питання захищеності, методів й підходів до реалізації захищених веб-застосунків є актуальним, як ніколи.

Об’єктом дослідження є процес аналізу найпоширеніших вразливостей безпеки веб-систем й розробка захищеної системи розумного будинку на основі проведеного аналізу.

Предметом дослідження є засоби й методи розробки захищеної системи розумного будинку на базі клієнт-серверного веб-застосунку.

Метою роботи є дослідження вразливостей безпеки, розгляд найпоширеніших вразливостей веб-застосунків та їх типів, визначення актуальних способів захисту системи, розробка захищеної системи розумного будинку, а також її подальше тестування.

Головними задачами при виконанні кваліфікаційної роботи стали наступні пункти:

- дослідження теми веб-застосунків, їх типової архітектурної моделі, основних типів та принципи побудови таких систем;
- огляд найпоширеніших вразливостей веб-застосунків, їх категорій, особливостей впровадження та негативний вплив на систему в цілому й на дані зокрема;
- розгляд методів забезпечення безпеки, боротьби з найпоширенішими критичними вразливостями веб-систем й правила побудови механізмів захисту веб-застосунків;

- проведення проектування розроблюваної системи, в ході якого необхідно визначити вимоги до веб-застосунку, зокрема до механізмів захисту й забезпечення безпеки від критичних вразливостей й реалізувати UML діаграми;
- вибір технологій для створення клієнту, серверу й бази даних захищеної системи розумного будинку, які дозволять реалізувати кращі підходи для забезпечення безпеки веб-застосунку від найпоширеніших вразливостей, згідно з вимогами до системи;
- створення системи розумного будинку з урахуванням сучасних підходів до реалізації механізмів захисту й методів боротьби з найпоширенішими вразливостями;
- опис реалізованих механізмів захисту й методів боротьби з вразливостями розробленої системи розумного будинку у вигляди клієнт-серверного веб-застосунку;
- тестування розробленої системи з використанням різних підходів до тестування, серед яких мають бути як мануальне тестування й автоматизовані види тестування, так і тестування з використанням сканеру коду;
- вироблення висновків та надання рекомендацій, щодо застосування технологій ReactJS, NodeJS й реляційної бази даних MySQL для створення системи розумного будинку, з використанням фундаментальних механізмів захисту й методів забезпечення безпеки застосунку від найпоширеніших вразливостей.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Веб-застосунки

Поняття «веб-застосунок» потребує деякого осмислення розвитку мережевих технологій, а саме проведемо дистинкцію між веб-сторінкою, веб-сайтом та веб-застосунком. Саме просте та базове з цих понять – це веб-сторінка, яка являє собою документ, відповідаючий певній структурі та описаний на мові розмітки гіпертексту або HTML (HyperText Markup Language), при чому доступ до цього документу можна отримати за допомогою деякої мережі (наприклад, всесвітньої). Серед контенту веб-сторінки можуть бути наявні посилання на інші веб-сторінки, а сукупність цих веб-сторінок, маючих такий логічний зв’язок, називають веб-сайтом.

При цьому, це поняття припускає наявність деякого смислового та стильового зв’язку між всіма веб-сторінками, щоб читач цього веб-сайту отримував відчуття того, що він користується єдиним продуктом. Окрім візуальних питань, веб-сайт торкається питань архітектури, оскільки зростання обсягу контенту на кожній веб-сторінці та загальної кількості пов’язаних веб-сторінок уповільнює швидкість завантаження сторінки та ускладнює навігацію по сайту, що може бути критичним для вибору певного веб-сайту користувачем.

Головною відмінністю веб-застосунку є наявність інтерактивності: веб-сайти лише представляють деяку статичну інформацію, в той час коли веб-застосунки змінюють свою поведінку та контент в залежності від користувача та його запитів. Тобто веб-застосунок – це динамічний веб-сайт, що уможливлює взаємодію з сервером, який обробляє запити та генерує відповідні веб-сторінки, які надсилає клієнту.

1.2 Клієнт-серверна архітектура

Всі веб-застосунки базуються на клієнт-серверній архітектурі, тому розглянемо її детальніше. В контексті веб-застосунків в якості клієнтів виступають веб-браузери, за допомогою яких користувач отримує

інтерпретовані веб-сторінки, що зручні для перегляду, а також формує та посилає запити до сервера, в якості якого виступає веб-сервер.

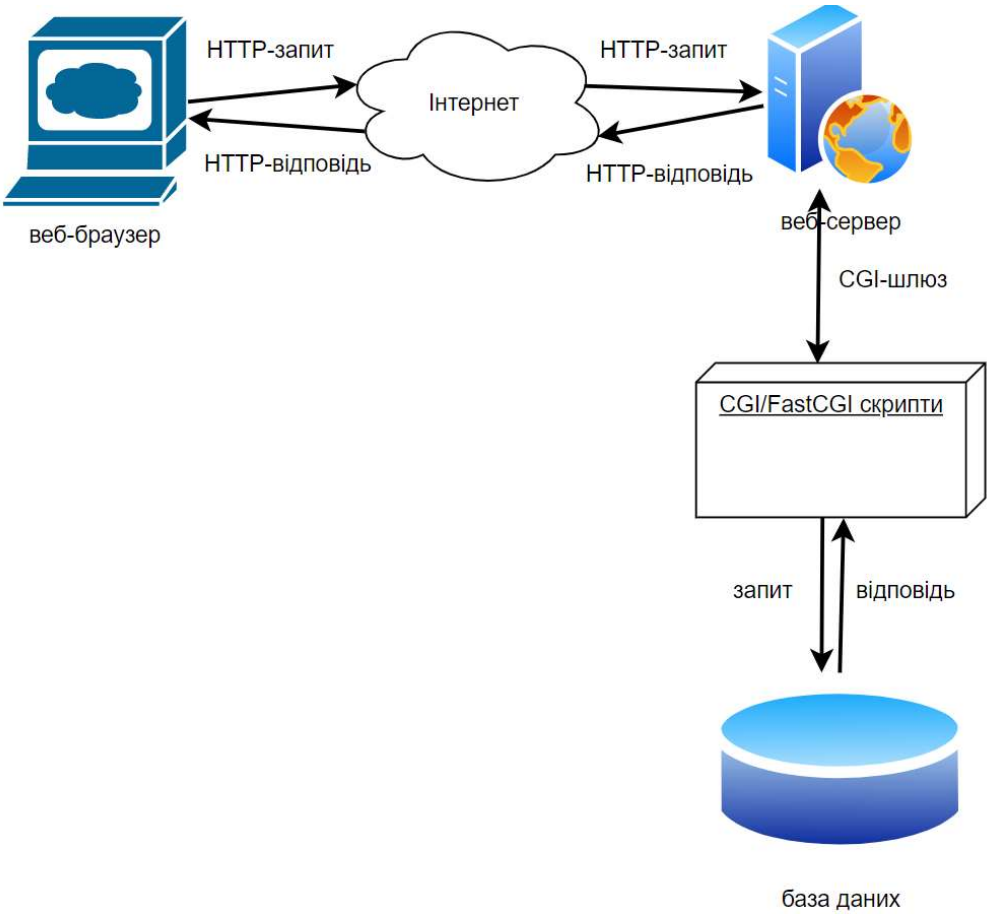

Рисунок 1.1 – Модель взаємодії у веб-застосунку

Для обробки інтерактивних запитів використовуються так звані CGI-скрипти (Common Gateway Interface). Насправді, CGI – це лише один з засобів динамічного формування веб-сторінок, це такий стандарт інтерфейсу, який дозволяє за допомогою будь-якої мови програмування, що має стандартні засоби вводу-виводу, організовувати взаємодію між клієнтом та сервером, дозволяючи звертатись до баз даних (БД), документів файлової системи та інших програм, обробляючи дані та формуючи відповідь. Окрім CGI можуть використовуватись і інші засоби генерації веб-сторінок, наприклад, FastCGI (який є розвитком CGI, більш швидкий та безпечний), PHP (PHP: Hypertext Preprocessor), JSP (Jakarta Server Pages), ASP (Active Server Pages) та інші.

Описані засоби використовуються за потребою: якщо від клієнта поступає запит якоїсь звичайної статичної веб-сторінки, то нема сенсу виконувати для

цього скрипт, але якщо треба здійснити обробку клієнтських даних, то без скриптів не обійтись.

З вищезазначеного стає зрозумілим, що серйозні веб-застосунки зазвичай використовують трирівневу клієнт-серверну модель з тонким клієнтом, тобто йде розділення клієнтського рівня, програмного та управління даними, а вся обробка даних та виконання застосунку забезпечуються сервером. З очевидних недоліків такого рішення є покладання всієї роботи на сервер, через що у разі його відмови стає неможливим робота застосунку, при чому відмова може бути як через поломку програмної чи апаратної частин, так й через зовнішні кібератаки, зокрема DDoS (distributed denial of service), від якого страждають навіть великі корпорації з величезними ресурсами.

1.3 Типи веб-застосунків

Класифікацію веб-застосунків можна проводити різними способами, але у цьому підрозділі виділяються типи з точки зору архітектури, при цьому не кожен з цих типів є виключним відносно інших.

1.3.1 Односторінкові веб-застосунки

Single-page application (SPA) – це такі застосунки, виконання яких проводиться у межах однієї веб-сторінки без її перезавантаження. Вони імітують стандартні десктопні застосунки та характеризуються тим, що працюють дуже швидко та плавно, а також не вимагають перезавантажень сторінки [1].

Такі сторінки розділяють розмітку та дані, дозволяючи робити окремі незалежні запити, після яких проводиться відповідне відображення отриманих даних прямо у браузері. Серед найвідоміших прикладів можна привести сервіси від Google, зокрема Gmail, який не оновлює сторінку цілком, перетворюючи її під потреби користувача.

Серед переваг можна виділити:

- швидкість роботи – зумовлюється тим, що майже всі необхідні ресурси завантажуються один раз, а в процесі роботи підвантажуються тільки дані (корисне навантаження);

- спрощується розробка, оскільки немає необхідності рендерити веб-сторінки на сервері;
- SPA-застосунок можна кешувати, після чого використовувати без доступу до мережи.

При цьому існують й недоліки односторінкових веб-застосунків:

- довге завантаження: оскільки для роботи відразу завантажуються всі необхідні ресурси, окрім корисного навантаження, то й перше завантаження відбувається довше;
- односторінкові веб-застосунки менш безпечні, ніж традиційні: міжсайтовий скриптинг або XSS (cross-site scripting) дозволяє зловмисникам робити ін’єкції шкідливого коду в застосунки інших користувачів.

1.3.2 Багатосторінкові веб-застосунки

Multiple-page application (MPA) – це протилежність SPA, для проведення змін у веб-браузері на стороні сервера формуються нові веб-сторінки, які надсилаються клієнтові. Зазвичай, багатосторінкові застосунки більше, ніж односторінкові, й представляють собою деякі великі продукти, як онлайн-магазини, соціальні мережі та інше. Однак за допомогою AJAX (Asynchronous JavaScript And XML) перетворення веб-сторінки не завжди потребує генерацію цілком нової веб-сторінки, уможливлюючи завантаження тільки необхідної інформації та відповідне оновлення контенту, проте все це ускладнює розробку та структуру веб-застосунку.

Отже, можна виділити наступні переваги:

- краща SEO (Search engine optimization), що пояснюється тим, що для кожної сторінки можна виділити окремі ключові слова, які підвищують певну веб-сторінку у пошуковій видачі;
- масштабованість – великі застосунки з багатим функціоналом та великою кількістю контенту неможливо реалізувати як SPA, оскільки в цьому випадку зручність його використання буде низькою. В той самий

час нічого не заважає розширити MPA, додавши пункт до меню навігації застосунку.

Але маються й недоліки:

- швидкість роботи менша за SPA, оскільки при переході на іншу веб-сторінку необхідне завантаження відповідного документу з веб-серверу;
- розробка фронтенду та бекенду сильно пов’язані, через що паралельна розробка цих частин неможлива;
- для підтримки працездатності веб-застосунку необхідно витрачати багато ресурсів.

1.3.3 Монолітні веб-застосунки

Під монолітними застосунками ми й звикли розуміти програми загалом, це єдиний застосунок, що вміщує в собі і інтерфейс, і доступ до даних, і їх обробку. Як правило, такі застосунки містять багато коду, обсяг якого з часом зростає, а підтримка ускладнюється. Це зумовлюється тим, що зміна коду якогось компоненту, потягне за собою потребу у зміні пов’язаних з ним компонентів, а зв’язки між компонентами у монолітних застосунках є тісними. Все це уповільнює процес розробки та модифікації застосунку.

Незважаючи на вищесказане, монолітні веб-застосунки є гарним вибором в разі розробки невеликих веб-застосунків через наступні переваги:

- просте розгортання веб-застосунку, оскільки є один виконуваний файл;
- для виконання витрачається менше ресурсів, оскільки відсутні витрати, які визиває наявність декількох відокремлених частин/модулів веб-застосунку, зокрема витрати на зв’язок між цими частинами;
- тестування веб-застосунку є комплексним, оскільки тестується єдина система;
- монолітні застосунки мають більш високу пропускну здатність.

Також виділимо описані недоліки:

- складний процес модифікації компонентів застосунку через сильну пов’язаність цих компонентів з іншими;
- недоцільність застосування підходу для великих веб-застосунків;
- монолітні застосунки також можуть бути складними для розуміння через велику кількість пов’язаного коду, що також ускладнює його модифікацію;
- при кожному оновлені коду, навіть однієї строки з безлічі, необхідно проводити повну перекомпіляцію коду та заново розгортати веб-застосунок;
- подібно до попереднього пункту, надійність також встає під питання, оскільки баг в якоїсь невеличкій частині програми зумовлює неправильну роботу всього застосунку, що може призвести до його зупинки.

1.3.4 Мікросервісні веб-застосунки

Ідея мікросервісної архітектури, яка протиставляється до монолітної, полягає у створенні великої кількості блоків коду, кожен з яких реалізує окремий сервіс. А загальний веб-застосунок складається з цих невеликих сервісів. Замість одного великого є декілька невеликих застосунків, кожен з яких може працювати незалежно від інших, при цьому вони можуть бути написані на різних мовах програмування та з використанням різних технологій. Відповідно й процес розробки може бути організовано окремими групами розробників, що робить процес розробки гнучкішим.

Зазвичай, для кожного сервісу створюється окремий контейнер, що має відокремлене від інших сервісів середовище виконання, а також окрему БД для цього мікросервісу, це зводить ймовірність впливу збоїв та помилок на інші сервіси до нуля.

Виділимо головні переваги мікросервісного підходу:

- модифікації коду можуть бути легко внесені та протестовані, тому що кожен з сервісів є невеликим за обсягом та незалежним від інших. При цьому зміни не вимагають повторної компіляції та повторного розгортання всього застосунку, а лише окремого сервісу;
- модульність застосунку дозволяє окремій групі розробників фокусуватись на певній темі та цілі сервісу, не відволікаючись на всі аспекти веб-застосунку;
- розуміння коду та процес розробки швидші та простіші за монолітні застосунки: це є наслідком попередніх пунктів;
- більш простий процес підтримки веб-застосунку у порівнянні з монолітними;
- більша високий рівень надійності: збій окремого сервісу не призводить до збою всього веб-застосунку;
- легший процес масштабування системи: якщо в монолітній системі не вистачає ресурсу, то необхідно розгортати ще один екземпляр всієї системи, в той час як в мікросервісній архітектурі можна розгорнути певний сервіс.

Проте мікросервісна архітектура має й недоліки:

- для організації зв’язку між сервісами витрачаються додаткові ресурси, так само як й для розгортання багатьох екземплярів застосунку (через створення відокремлених контейнерів);
- тестування окремих сервісів простіше, але тестування всього застосунку, тобто взаємодії сервісів, складніше, ніж у монолітній системі;
- мікросервісний застосунок необхідно детально аналізувати з точки зору безпеки, оскільки неналежне управління привілеями може привести до критичних вразливостей;

- необхідне дотримання узгодження даних, оскільки кожен сервіс має окрему базу даних.

1.3.5 Безсерверні веб-застосунки

Незважаючи на назву, ці веб-додатки також мають клієнт-серверну архітектуру, проте обов'язки сервера покладаються на третю сторону – деякого хмарного провайдера [2]. Тобто розробники концентруються на програмній стороні продукту, не піклуючись про апаратне забезпечення та його налаштування.

При цьому це може бути як PaaS (Platform as a Service), коли розробники отримують готову для використання платформу, на яку необхідно завантажити застосунок для запуску, так і FaaS (Function as a Service), коли завантажуються певні функції, а для їх запуску створюються певні тригери подій, наприклад, надходження HTTP-запиту або електронної пошти.

Переваги такого підходу:

- простота розгортання;
- спрощений процес підтримки застосунку;
- відпадає необхідність закупівлі дорогої техніки, а провайдери стягують гроші тільки під час виконання застосунку або за виклик окремої функції;
- питання масштабування застосунку покладено на провайдера.

Хоча процес розробки сильно спрощується, такий підхід має й відповідні недоліки:

- знижена безпека даних: окрім того, що ви надаєте дані третій стороні, на одному сервері можуть виконуватися та зберігатися різні застосунки, що при неналежному контролі доступу та наявних вразливостях може призвести до витоку інформації;
- для виклику функції при спрацьовуванні тригеру існує затримка;
- складність проведення повного тестування застосунку з врахуванням інтерактивності окремих функцій;

- неможливість впливу на збої обладнання;
- можливість використання лише тих технологій, що підтримує провайдер.

1.3.6 Прогресивні веб-застосунки

Progressive web application (PWA) – це такий вид веб-застосунків, що можуть використовуватись одночасно як веб-застосунок у браузері, так і застосунок на смартфонах та навіть настільних пристроях (завдяки браузерам, підтримуючим цю технологію). Розробляються вони, подібно до веб-застосунків, за допомогою HTML, CSS (cascading style sheets), JavaScript, але при цьому мають властивості мобільних додатків: працюють швидко та в режимі офлайн, використовувати можливості девайсів, наприклад, можуть надсилати повідомлення [3]. При цьому для того, щоб вважати веб-застосунок PWA, необхідно виконання декількох умов: використання захищеного протоколу HTTPS (HyperText Transfer Protocol Secure); наявність service worker'у – це такий шар між веб-браузером та Інтернет-мережею, по суті, це скрипти, що керують HTTP-запитами; наявність файлу маніфесту – документ у форматі JSON (JavaScript Object Notation), який містить метадані веб-застосунку.

Виділимо переваги такого підходу:

- розробка з використанням стандартних веб-технологій дозволяє одночасно створювати мобільний застосунок, заощаджуючи ресурси;
- завантажені застосунки не вимагають оновлення;
- кросплатформність;
- можливість роботи без наявності Інтернет-з'єднання.

Попри описані вище переваги, такий тип веб-застосунків містить наступні недоліки:

- більш високе споживання ресурсів системи;
- має менше можливостей використання функцій пристрою, на відміну від повноцінних додатків, що встановлюються на пристрій (наприклад, неможливість використання Bluetooth);

- попри високу ефективність у порівнянні з звичайними веб-застосунками, PWA поступаються стандартним застосункам, що завантажуються на пристрої.

1.4 Вразливості веб-застосунків

Для розуміння того, як робити захищені веб-застосунки, необхідне детальне розуміння загроз, направлених на продукт. Для визначення найбільш актуальних вразливостей веб-застосунків скористуємось найпопулярнішим джерелом – топом OWASP Top 10 (Open Web Application Security Project) від 2021 року [4]. Кожного року поширеність вразливостей певних категорій змінюється, а нові атаки продовжують створюватися, тому необхідно приділяти увагу сучасному стану речей та проводити періодичний аналіз сфери безпеки веб-застосунків. Наприклад, за 4 роки деякі категорії вразливостей стали більш актуальними, з'явились нові категорії, що в свою чергу враховують вразливості тих категорій, що зникли з топу, ці зміни можна побачити на рисунку 1.2, який приводять автори рейтингу.

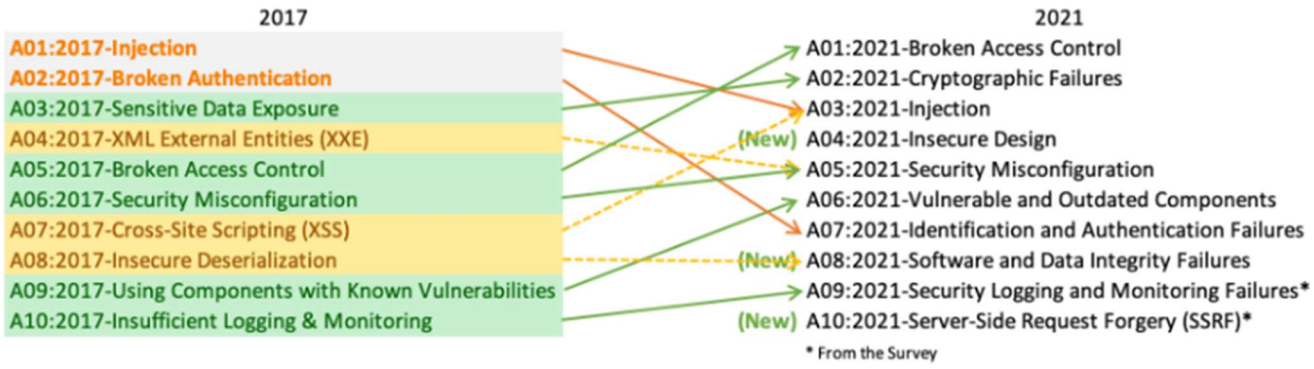

Рисунок 1.2 – Зміна рейтингу вразливостей

У цьому топі визначені великі категорії, кожна з яких може включати багато різних вразливостей, завдяки чому ми можемо бачити тенденції розвитку та напрямку кібератак, розуміючи, на що треба звернути додаткову увагу при розробці. Отже, розглянемо визначені категорії детальніше за порядком появи у топі 2021 року.

1) Порушення контролю доступу

Впровадження контролю доступу має мету обмежити доступ певних користувачів до ресурсів, до яких вони доступ мати не повинні, чи то в умовах одного пристрою зберігання даних, чи до мережевих ресурсів. Окрім доступу до ресурсів, певні користувачі (або групи користувачів) повинні мати змогу проводити тільки дозволені їм операції. Наприклад, звичайні користувачі не можуть видаляти документи в директоріях, що були створені не ними. Порушення такого контролю зазвичай призводять до порушення конфіденційності та цілісності інформації.

2) Помилки криптографії

Передбачається використання небезпечних або застарілих алгоритмів, також можливе використання слабких параметрів для цілком ефективних та сучасних алгоритмів (наприклад, маленька довжини ключа або певні слабкі ключі, що не рекомендуються для використання). Також можлива передача конфіденційних даних з використанням протоколів зв'язку, які зовсім не забезпечують шифрування – HTTP, FTP (File Transfer Protocol) та подібні [5].

3) Ін'єкції коду

Завдяки таким вразливостям зловмисник може передавати на виконання шкідливий код через вразливий веб-застосунок іншим клієнтам та на сервер, тобто в інші системи. Причому варіація ін'єкцій досить велика: це можуть бути SQL та NoSQL ін'єкції, ін'єкції, що використовують системні команди (тобто операційної системи), одна з найнебезпечніших – XSS, яка вже згадувалась раніше, та інші.

В якості запобігання можна створити прикладний програмний інтерфейс замість прямої передачі запитів в інтерпретатор, робити додаткові перевірки запитів.

4) Небезпечна структура

Ця категорія передбачає вразливості, що з'явились внаслідок некоректного процесу проектування веб-застосунку, зокрема використання невдалої

архітектури для системи. Щоб запобігти таким вразливостям, рекомендується використовувати принципи побудови еталонних архітектур, а також шаблони проектування. Також корисним буде моделювання загроз веб-застосунку (при чому не тільки для цієї категорії, а в цілому).

5) Порушення конфігурації безпеки

Як зрозуміло з назви, ця категорія описує вразливості, що виникають при неналежному налаштовуванні безпеки веб-застосунка. Наприклад, це можуть бути відкриті порти, що не потрібні для використання цього застосунку, наявність облікових записів за замовчуванням, за допомогою яких можна авторизуватись в системі, параметри безпеки окремих компонентів системи не налаштовані належним чином.

6) Вразливі та застарілі компоненти

Суть цієї категорії також зрозуміла з назви: описуються вразливості, виникаючі внаслідок використання неоновлених бібліотек, СКБД (системи керування базами даних) та інші інструменти, а також ігнорування оновлень операційної системи. Також можна віднести випадки, коли система продовжує використання компонентів, інформація про зараження яких була опублікована.

7) Проблеми з аутентифікацією та ідентифікацією

Ця категорія є досить обширною та описує всі вразливості, що пов’язані з процесом аутентифікації. Хоча більшість розробників веб-застосунків знає про важливість організації цього процесу та зміну позиції категорії у рейтингу з другого до сьомого, проблема все ще актуальна й понині. Серед головних вразливостей можна виділити відсутність перевірок паролів та дозвіл користувачам встановлювати слабкі та відомі паролі, які не відповідають сучасним вимогам, відсутність багатофакторної аутентифікації. Також можна віднести вразливості, що дозволяють зловмисникам використовувати автоматизовані атаки з перебору паролів в веб-застосунку. Окрім прямого процесу аутентифікації, категорії належать вразливості, завдяки яким можна вилучити ідентифікатори сеансу (наприклад, з URL (Uniform Resource Locator)).

8) Порушення цілісності даних та програмного забезпечення

Вразливості цієї категорії засновані на завантаженні бібліотек та модулів для використання у веб-застосунку з ненадійних джерел, тобто таких, що можуть містити шкідливий код. Це може привести як к витоку даних, так і компрометації всієї системи. Для запобігання цим вразливостям треба використовувати тільки перевірені та ліцензовані версії бібліотек та залежностей, завантажених з офіційних ресурсів.

9) Помилки реєстрації та моніторингу безпеки

Ця категорія описує вразливості, що наявні внаслідок неналежного ведення моніторингу та реєстрації подій, пов’язаних з безпекою застосунку. Через ці вразливості стає неможливим не тільки виявити причину, але й зрозуміти, що з системою щось не так. Окрім відсутності реєстрації, існує проблема неналежної реєстрації, коли повідомлення малоінформативні та з них не можна однозначно встановити, що та чому трапилось. Не менш важливим є й резервне копіювання журналів реєстрації.

10) Підробка запитів з боку сервера

Завдяки вразливостям цієї категорії зловмисник може заставити веб-сервер звернутись до деякого посилання, наприклад, якоїсь зовнішньої системи в мережі. Внаслідок цього можливий виток конфіденційних даних користувачів. Або зловмисник може змусити сервер відправляти HTTP-відповіді з локальною інформацією машини, посилаючись на URL-адресу зворотного зв'язку. Для запобігання цим атакам можна фільтрувати весь трафік, окрім необхідного, вимкнути можливість перенаправлення. Також можна зробити додаткові перевірки запитів від клієнтів.

1.5 Безпека веб-застосунків

Забезпечення безпеки веб-застосунків – це комплексний, складний та дорогий процес, яким проте не нехтують жодні розробники якісного ПЗ (програмного забезпечення), оскільки збої системи та витоки даних користувачів можуть не тільки нанести відчутних грошових збитків, але й зіпсувати

репутацію, внаслідок якої користувачі відмовляться від використання застосунку (проте, це теж зводиться до грошових збитків). Процес цей є многогранним, тому що безпека веб-застосунку залежить ще від процесу його проектування, тому що, як зрозуміло з підрозділів 1.3 та 1.4, обрана архітектура застосунку впливає на його безпеку самим прямим чином. Так само як і використовувані в розробці технології та сторонні бібліотеки, що використовуються для надання певного функціоналу.

Проте зрозуміло, що масштаб процедур забезпечення безпеки напряму залежить від масштабу веб-застосунку та використаним при його розробці технологій. В цьому підрозділі розглядаються різні аспекти створення безпечного та надійного веб-застосунку та конкретні заходи, які необхідно проводити для підтримки безпеки на протязі всього життєвого циклу. Деякі з визначених нижче аспектів випливають напряму з категорій вразливостей, розглянутих у попередньому підрозділі.

1.5.1 Безпечна архітектура веб-застосунку

Вчасний та належний аналіз архітектури застосунку є одним з найважливіших етапів розробки, оскільки без цього етапу можуть з’явитись проблеми під час написання коду та навіть експлуатації, виправити які буде неможливо без відповідних архітектурних модифікацій.

Важко надати детальні правила та рекомендації, оскільки багато рішень залежать від бізнес-вимог веб-застосунку та обраного типу веб-застосунку, проте можна виділити найголовніші та загальні положення [6]:

- при можливості бажано мати окремі сервери для виконання застосунку та для зберігання даних, а також мати резервні сервери, що можуть замінити основний у разі відмови та збоїв;
- обмеження можливості вводу від користувачів там, де це не є необхідним, а також перевірка користувацького вводу на стороні сервера, можлива також перевірка на стороні користувача при необхідності, це зменшує навантаження на сервер;

- використання засобів зв’язку з шифруванням (HTTPS замість HTTP), відмова від можливості використання застарілих версій протоколів, наприклад, станом на 2022 рік застарілими версіями протоколів, що не рекомендуються до використання, визнані всі версії SSL (Secure Sockets Layer) та версії TLS (Transport Layer Security) нижче 1.2;
- забезпечення підтримки багатофакторної аутентифікації у веб-застосунку;
- зберігання облікових даних, що використовуються користувачами для входу, повинно проводитись виключно у зашифрованому стані, при чому паролі потрібно подавати на вхід геш-функції, а не звичайного шифру, оскільки геш має функцію незворотності: з обчисленого геш-значення неможливо отримати вихідне значення;
- хоча напряму це не пов’язано з розробкою веб-застосунку, проте на нього впливає й наявність мережевих фільтрів та систем виявлення або запобігання вторгнень, що підвищують надійність роботи веб-застосунку та його стійкість до відмов.

1.5.2 Аналіз коду

Весь час розробки необхідно проводити аналіз коду та спостерігати за тим, звідки бібліотеки завантажуються та що це за бібліотеки в цілому. Навіть якщо автор бібліотеки не є зловмисником, її код може бути шкідливим та вразливим.

Такий аналіз коду передбачає використання автоматизованих сканерів для детектування вразливостей та підозрілих сторонніх бібліотек, а саме використовується методологія, що називається SAST (Static Application Security Testing), вона передбачає аналіз вихідного коду для знаходження вразливостей, для позначення методу також використовується тестування «білої скриньки» [7].

Цей підхід використовується з самого початку розробки веб-застосунки після етапу аналізу архітектури системи та допомагає виправляти помилки на самому початку їх уникнення. Проте лише проведення таких сканів не є достатнім, оскільки можливі як хибно позитивні спрацьовування, так і не

виявлення вразливостей, що присутні у коді. Тому бажано мати спеціаліста з кібербезпеки, який буде проводити аналіз результатів та приймати рішення щодо усунення наявних вразливостей. Окрім цього, сканер, як правило, визначає рівень критичності знайденої загрози, що також допоможе в прийняті рішення про пріоритет по усуненню окремих вразливостей.

1.5.3 Проведення оцінки ризиків

Оцінка ризиків – це ітеративний та комплексний процес, який є дуже важливим при розробці та підтримці веб-застосунку та може бути досить варіативним, оскільки існує досить багато підходів та методологій. Зазвичай управління ризиками включає в себе етапи встановлення контексту проекту, ідентифікацію ризику (включно вразливостей та загроз), після чого йде визначення ймовірностей вразливостей та загроз, а також обчислення значення ризику за обраною методологією. Нарешті, маючи результати оцінки, приймаються рішення щодо пом’якшення тих чи інших ризиків в залежності від ступеню їх критичності та важливості активів, на які вони розповсюджуються.

Якщо в нашому випадку актив – це розроблюваний веб-застосунок, то для ідентифікації можна використовувати ти ж самі сканери. SAST був обговорений у попередньому пункті, але ці інструменти дозволяють виявити вразливості, що наявні у коді застосунку. Але не всі вразливості проявляють себе у коді, деякі можна виявити тільки при роботі застосунку, для чого і використовується DAST (Dynamic Application Security Testing), який також називають тестуванням «чорної скриньки». Суть полягає у тому, що сканер не має доступу до вихідного коду веб-застосунку, а пошук вразливостей зводиться до звичайного посилання HTTP-запитів серверу за допомогою фронтенду. Провівши такий аналіз, сканер здатен сформувати звіт з зазначенням знайдених вразливостей та, можливо місць у коді, де ці вразливості наявні (SAST завжди зазначає певне місце), та, в залежності від конкретного сканеру, визначають кроки по усуненню вразливостей.

Комбінування SAST та DAST дозволяє перекрити їхні мінуси та проводити аналіз якомога ефективніше та надійніше, проте існують більш нові методики тестування, одна з них називається IAST (Interactive Application Security Testing) та працює зсередини веб-застосунку, аналізуючи код та проводячи динамічний аналіз програми. При чому ефективність знаходження вища за DAST, а ймовірність хибно позитивних спрацьовувань значно нижча [8]. Хоча, на відміну SAST, таке інтерактивне тестування може знаходити вразливості під час роботи застосунку, це тестування покриває не весь вихідний код застосунку, а лише ті, з якими відбувається взаємодія користувача, а ефективність і точність менша за SAST. Можливо, це пов’язано з тим, що інструменти статичного тестування розробляються набагато довше за інструменти інтерактивного тестування, і з розвитком технології стан речей зміниться в кращу сторону. Ще одним недоліком застосування IAST є погіршення продуктивності веб-застосунку, оскільки такий аналіз досить відчутно впливає на швидкість роботи.

Найбільш новою технологією є RASP (Runtime Application Self Protection), що теж працює зсередини веб-застосунку, але сутність полягає у тому, що цей інструмент здатен не тільки виявляти вразливості, але й запобігати певним атакам при роботі веб-застосунку. Вперше RASP був визначений у [9], як технологія забезпечення безпеки, що вбудована пов’язана з застосунком або його середовищем виконання, та здатна контролювати виконання програми, а також виявляти атаки та запобігати їм у реальному часі. Зазвичай інструмент представляє собою модуль/бібліотеку що використовуються у застосунку, або агента, який спостерігає за роботою застосунка. Також можливий варіант, в якому RASP замінює собою віртуальну машину, що виконує функції забезпечення безпеки. Проте RASP складно порівнювати з вище розглянутими видами тестування, оскільки, хоча він і здатен запобігати атакам, його ефективність у знаходженні вразливостей не така висока. Це все може бути наслідком новизни технології та якості окремого інструменту, але при проведенні дослідження у [9] RASP-інструмент був здатен виявити тільки атаки,

що відносяться до категорії «Ін’єкція». При цьому зрозуміло, що подібно до IAST, RASP також сповільнює роботу веб-застосунку та споживає додаткові ресурси, при цьому він має певні права, щоб керувати роботою веб-застосунку для забезпечення його безпеки. Наприклад, він може припиняти сесії певних користувачів, завершати роботу застосунку, аналізувати потоки логіки та даних веб-застосунку.

1.5.4 Використання надійних та ефективних засобів шифрування

Окрім наявності шифрування, що є необхідним не тільки для зв’язку між сервером та клієнтами, але й для зберігання даних користувачів, у тому числі облікових даних, обраний алгоритм шифрування повинен задовольняти певним вимогам [10]:

1) знання алгоритму, за яким проводиться шифрування, не повинно надавати можливість розшифрування шифртексту, а безпека повинна визначатись секретним ключем;
2) відсутність слабких ключів, які дозволяють зламати систему ефективніше, ніж атака «грубої сили»;
3) розшифрування повідомлення може бути здійснено тільки за наявності секретного ключа;
4) алгоритм повинен підтримувати достатній рівень криптостійкості з урахуванням розвитку обчислювальних технологій;
5) процес шифрування та розшифрування повинен буду обчислювально простим за наявності секретного ключа;
6) підтримка властивості лавинового ефекту: при невеликій модифікації відкритого тексту, шифртекст повинен змінюватись значним чином, навіть при використанні одного і того ж самого ключа; а також при невеликій модифікації ключа, шифртекст повинен змінюватись значним чином, навіть при шифруванні одного і того ж відкритого тексту;

Також вимоги висуваються й до геш-функцій:

1) забезпечення незворотності: з геш-значення обчислювально неможливо отримати повідомлення, від якого воно було отримане;
2) стійкість до колізій першого роду: повинно бути обчислювально неможливо знайти таке повідомлення, що відрізняється від вихідного, але геш-значення якого співпадає з вихідним;
3) стійкість до колізій другого роду: повинно бути обчислювально неможливо знайти два повідомлення, геш-значення яких співпадають;
4) геш-функції також повинні відповідати лавиновому ефекту: незначна модифікація повідомлення значним чином змінює обчислювальне геш-значення.

1.5.5 Розробка механізмів моніторингу та реєстрації подій

Процес реєстрації подій або ведення логів є вкрай важливим для будь-якого застосунку, при чому не тільки звичайні логи під час розробки та підтримки застосунку, що дозволяють відслідковувати помилки та причини збоїв, але й логи безпеки, в яких йде запис про важливі системні події та події безпеки.

Переглядаючи логи, системні адміністратори можуть відслідковувати дії користувачів, а працівники, відповідаючи за безпеку, можуть розслідувати виявлені порушення. Зазвичай, логи фіксують наступні типи інформації [11]:

- назва події;
- опис події;
- мітка часу, коли подія відбулась;
- користувач або служба, що створили, змінили або видалили подію;
- застосунок або пристрій, на який вплинула подія (IP-адреса, ідентифікатор пристрою);
- джерело походження користувача або служби (країна, ім’я хосту, IP-адреса, ідентифікатор пристрою);
- важливість або серйозність події (може мати різну кількість рівнів).

Події, які, як правило, підпадають под логування:

- адміністративна активність (все, що пов’язано з роботою адміністраторів, наприклад, створення/видалення облікових записів користувачів);
- доступ до даних або їх модифікація (перегляд файлів, зміна вмісту файлів, завантаження, створення нових файлів);
- помилки при спробах аутентифікації в системі (неверний ввод вхідних даних облікового підпису);
- помилки при спробах доступу до файлів, до яких користувач не має прав;
- системні зміни.

1.5.6 Налаштування політики безпеки веб-застосунку

Політика безпеки веб-застосунку складається з декількох пунктів. Виділимо та опишемо основні з них.

1) Контроль доступу та авторизація

Цей пункт є одним з найголовніших з точки зору безпеки веб-застосунку, особливо приймаючи до уваги те, що в топі OWASP відповідна категорія вразливостей стоїть на першому місці.

Зазвичай гарно розроблений процес авторизації гарантує, що тільки авторизовані користувачі можуть виконувати дозволені для їх рівня привілей дії. В межі авторизації входить управління доступом до захищених ресурсів, а рішення про доступ приймаються або на основі ролі користувача, або на основі наявних саме у нього привілей. Раніше, як правило, для уможливлення контролю створювались списки контролю доступу або ACL (Access Control List), що визначали права доступу певних користувачів або ролей до конкретного об’єкту в системі. Але в веб-застосунках такий підхід не є актуальним, перевага віддається заснованому на ролях контролі доступу або RBAC (Role-Based Access Control), тому що він має наступні переваги:

- ефективність відносно інших підходів: для будь-якої групи, пов’язаною з веб-застосунком, будь то розробники, адміністратори або користувачі, можна створити окрему роль, яка буде складатись з мінімального обсягу привілей, що необхідні для їх роботи з застосунком. Завдяки такій можливості, процес надання та відібрання привілей сильно полегшується, причому не тільки для конкретного користувача, але й для всієї групи: можна в будь-який момент змінити привілеї самої ролі;
- завдяки використанню ролей стає легшим проводити аудит, оскільки набагато легше перевірити, які саме користувачі мають відповідну роль і, отже, відповідну привілею;
- високий рівень масштабованості, причому як горизонтальної – надання новому користувачу ролі, так і вертикальної – створення нової ролі для деякої групи користувачів.

Для розробки коректного процесу авторизації можна виділити наступні рекомендації:

- не використовуйте аутентифікацію на боку клієнту, а також засновану на токенах аутентифікацію, оскільки остання, хоча й ефективніша за використання сесій, має нижчий рівень безпеки. Завдяки аутентифікації на основі сесій, адміністратор серверу має змогу обірвати сесію у разі виникнення підозрілої активності;
- не використовуйте статичні посилання для завантаження файлів, а генеруйте їх динамічним чином. Якщо ж посилання повинно бути статичним, то обов’язково потрібно встановити контроль доступу до завантаження цього файлу;
- у веб-застосунку для перегляду користувачем сторінки або виконання якоїсь дії завжди повинна бути завчасно виконана перевірка дозволу, узгоджена з централізованим компонентом авторизації.

2) Парольна політика

Проблема використання слабких паролів є нескінченною, оскільки більшість користувачів віддають перевагу не безпечності та надійності паролю, а його простій запам’ятовуваності.

Існує багато політик безпеки, що визначені світовими організаціями з питань безпеки та стандартизації, але не всі вони відповідають сучасним вимогам, отже згенеруємо свої правила для гарної політики безпеки веб-застосунку:

- при трьох-п’яти невдалих поспіль спробах аутентифікації необхідно блокувати можливість аутентифікації, вимагаючи від користувача підтвердження його особистості за допомогою електронної пошти чи повідомленню на його номер телефону. Це правило є критичним, оскільки воно унеможливлює атаку «грубої сили»;
- вимагайте використання літер обох регістрів, а також хоча б однієї цифри та спеціального символу. Все це розширює можливий алфавіт, з якого складаються паролі, що ускладнює його злам;
- завдяки великому алфавіту, можна зменшити мінімальну кількість символів паролю, можна використовувати 8 символів, детальна статистика по швидкості зламу паролів наведена у таблиці 1.1;
- забороніть повторне використання минулих паролів користувача;
- перевіряйте паролі користувачів зі списком найпоширеніших та найслабкіших паролів та забороняйте можливість їх використання.

Наведемо актуальну таблицю швидкості зламу паролю з використанням сучасних обчислювальних пристроїв, розраховану у [12], що враховує розміри алфавіту, що використовується для його створення.

Таблиця 1.1 – Швидкість зламу паролю

Кількість символів	*Тільки цифри*	*Літери нижнього регістру*	*Літери обох регістрів*	*Цифри та літери обох регістрів*	*Цифри, літери обох регістрів та спеціальні символи*
6	Миттєво	1 хв.	1 год.	4 год.	15 год.
7	3 сек.	35 хв.	3 дні	2 тижні	2 місяці
8	26 сек.	15 год.	5 місяців	2 роки	10 років
9	4 хв.	2 тижні	23 роки	100 років	800 років
10	44 хв.	1 рік	10^3 років	$7*10^3$ років	$61*10^3$ років
11	7 год.	31 рік	$63*10^3$ років	$435*10^3$ років	$5*10^6$ років
12	3 дні	800 років	$3*10^6$ років	$27*10^6$ років	$363*10^6$ років
13	1 місяць	$21*10^3$ років	$170*10^6$ років	$2*10^9$ років	$28*10^9$ років
14	10 місяців	$539*10^3$ років	$9*10^9$ років	$104*10^9$ років	$2*10^{12}$ років
15	8 років	$14*10^3$ років	$460*10^9$ років	$6*10^{12}$ років	$166*10^{12}$ років
16	84 роки	$365*10^6$ років	$24*10^{12}$ років	$399*10^{12}$ років	$13*10^{15}$ років

1.5.7 Резервне копіювання даних

Резервне копіювання – це надзвичайно важливий процес для будь-якої системи, тим паче якщо веб-застосунок зберігає дані користувачів. Періодичність цього процесу залежить від швидкості змін даних у веб-застосунку, та може бути як щоденна (є рекомендованою), так і щотижнева. Втрати дані можна за різними причинами: будь то збої та поломки фізичного обладнання або кібератаки на систему.

Краще всього, якщо бюджет дозволяє робити декілька різних копій, що зберігаються на різних платформах, наприклад, на зовнішньому накопичувачі, на хмарному сервері якогось провайдера. Якщо коштів недостатньо, то слід віддати перевагу хмарному серверу, що не постраждає від атаки на веб-застосунок або від проблем в серверній вашого веб-застосунку.

Потрібно визначити час, який резервна копія повинна зберігатись: дисковий простір не є безмежним, а тим паче безкоштовним, тому кожна резервна копія повинна мати час зберігання, після якого вона видаляється, звільняючи простір для нових копій даних.

2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ЗАСТОСУНКУ

У цьому розділі визначаються різні вимоги до розроблюваної системи, серед яких можна виділити наступні: функціональні вимоги, вимоги до інтерфейсу, вимоги до серверу, вимоги до механізмів захисту. На основі визначених вимог відбувається проектування системи в цілому й обираються технології розробки системи. Після розробки системи розглядаються розроблені компоненти системи й механізми захисту системи.

2.1 Визначення функціональних вимог до системи

На основі проведеного мною дослідження предметної області, пошуку та аналізу існуючих програмних рішень та ознайомлення з типовим архітектурами веб-застосунків були визначені певні вимоги до розроблюваного застосунку.

Прийнято рішення розробити веб-застосунок на базі клієнт-серверної архітектури, котрий відповідав би всім сучасним стандартам й методам безпеки [13]. В той же час система має відповідати нормам забезпечення належної безпеки, але бути простої, інтуїтивно зрозумілою та зручною для користування юзером.

Серед функціональних вимог до додатку можна виділити наступні:

- розроблена веб-програма має коректно працювати й відображатися без візуальних критичних помилок помилок у любому веб-браузері та на будь-якій операційній системі;
- юзеру має бути доступна опція реєстрації нового акаунту;
- юзеру має бути доступна опція використання створеного акаунту для роботи у додатку;
- юзеру має бути доступна можливість вийти зі свого акаунту;
- юзеру має бути доступна можливість додати новий пристрій для себе;
- кожен пристрій має бути здатним мати назву, тип, режими роботи, зображення;

- юзеру має бути доступна можливість проглядати свої існуючі пристрої у будинку;
- юзеру має бути доступна можливість редагувати існуючі пристрої;
- юзеру має бути доступна можливість видаляти пристрої;
- юзер повинен мати змогу змінювати змінювати статус, характеристики що залежать від типу пристрою, режим роботи пристрою;
- кожен пристрій повинен мати навігацію для управління;
- юзер повинен мати змогу сортувати пристрої за їх типом;
- юзер повинен мати можливість здійснювати пошук пристрою за назвою пристрою.

2.2 Визначення вимог до клієнту програми

Інтерфейс розроблюваної системи розумного будинку повинен відповідати належним чином сучасним поняттям простоти й бути ергономічним, що дозволить любому користувачу користуватися веб-додатком [14].

У клієнтській частині веб-додатку, а саме у візуальному інтерфейсі програми мають бути реалізовані наступні особливості:

- сторінка з полями створення нового облікового запису юзера повинна мати відповідні поля для введення логіну, паролю та кнопку створення облікового запису;
- сторінка входу до облікового запису юзера має включати в себе поля для введення логіну, паролю та кнопку ініціювання процесу входу до облікового запису;
- на головній сторінці мають відображатися пристрої усіх типів в будинку;
- на головному екрані має знаходитися кнопка додавання нового пристрою, котра відкриває форму додавання нового пристрою;
- форма додавання задачі включає в себе вибір типу пристрою, назви пристрою, поля додавання зображення пристрою, режимів роботи

пристрою та додаткових характеристик пристрою в залежності від типу пристрою;

- має бути реалізовано кнопки відміни, збереження та повернення на крок назад у формі створення та редагування пристрою;
- на головній сторінці має бути поле пошуку пристрою за назвою;
- на головній сторінці мають бути кнопки для сортування пристроїв за типом;
- при виборі пристрою має відкриватися форма редагування пристрою;
- форма редагування пристрою має включати в себе кнопку включення або виключення пристрою, розгортаючийся список з вибором режиму роботи пристрою;
- мають бути реалізовані кнопки видалення пристрою, кнопки збереження оновленої інформації про пристрій, кнопки повернення назад на головну сторінку застосунку та функціоналу по роботі з пристроєм в залежності від типи пристрою.

2.3 Визначення вимог до серверу програми

На веб-сервері програми має бути передбачено те, що веб-додаток одночасно можуть використовувати одразу декілька клієнтів, а саме два і більше. Саме по цій причині дуже важливо, щоб від був гарно масштабований для використання у режимі декількох клієнтів. Також передбачається , що реалізований сервер буде працювати з БД та їх взаємодія буде в достатній мірі оптимізована.

Також, головною необхідністю є те, щоб розроблена серверна частина застосунку працювала за Representational State Transfer (REST) принципами [15]. Працюючи за принципами REST після надсилання від клієнта на сервер останній не бути запам’ятовувати у собі той стан, у якому перебуває клієнтська частина програми.

Також, важливим моментом є те, що у свою чергу клієнту має бути недоступною інформація про те, чи є будь-яка точка, що обробляє його запит

кінцевою чи ні. У разі дотримання таких принципів ні один із компонентів розробленої системи не буде знати про наявність наступного. Попри це, кожен наступний компонент системи буде мати змогу отримати інформацію про свого попередника. Системна інформація та дані мають використовувати такі формати передачі даних, як JSON чи XML (Extensible Markup Language).

На сервері має бути реалізований деякий API (Application Programming Interface), котрий буде обробляти запити, що надсилає клієнтська частина застосунку до розробленого сервера. Тобто, сервер повинен мати змогу здійснювати обробку таких HTTP запитів, як: PATCH, DELETE, POST, GET та запити PUT [15].

На сервері мають бути реалізовані деякі функції, а саме:

- створення нового облікового запису юзера;
- вхід юзера до свого облікового запису;
- вихід юзера зі свого облікового запису;
- створення нового будинку;
- отримання всіх будинків;
- отримання будинку;
- редагування будинку;
- видалення будинку;
- створення нового пристрою;
- отримання пристрою;
- редагування пристрою;
- видалення пристрою;
- пошук пристрою за назвою.

2.4 Визначення вимог до механізмів захисту системи

В ході проектування системи було визначено деякі вимоги до механізмів захисту веб-застосунку, серед яких можна виділити наступні [16]:

- Мають бути реалізовані валідації, фільтрації та перевірки всіх видів вхідних даних користувача;

- Неавторизований користувач чи зловмисник повинен не мати змоги перехопити дані веб-застосунку, які відправляються з клієнта чи повертаються на клієнт;
- Забезпечення санітайзингу вхідних даних;
- Передача даних має забезпечуватися безпечними методами зв’язку;
- Мають бути використані методи хешування паролів, задля забезпечення безпеки даних. Тобто у БД має зберігатися саме хеш, замість паролю й при кожній спробі авторизації користувача порівнюватися має саме хеш введеного паролю, а не сам пароль;
- Має бути реалізований алгоритм автентифікації користувачів до ресурсів з використанням токенів;
- Забезпечення методу підтвердження дій користувача, завдяки надсиланню листів на його електронну пошту;
- Має бути забезпечено спроможність блокувати багаторазові спроби входу до облікового запису користувача;
- При створенні нового облікового запису користувача, мають бути застосовані вимоги щодо складності створюваного паролю, що зменшить ймовірність підбору паролю користувача;
- Використовувати формат передачі даних JSON, замість формату XML;
- Застосування інструментів сканування методом SAST, задля забезпечення якості програмного коду зокрема, й системи в цілому й виявлення вразливостей.

2.5 Проектування системи

Згідно з визначеними раніше вимогами до розробки системи розумного будинку, які були оговореними в 2 розділі в пунктах 2.1, 2.2, 2.3 було розроблено наступні діаграми:

- use-case діаграма;
- діаграма послідовності створення нового облікового запису юзера;
- діаграма послідовності автентифікації юзера;

- ER-діаграма.

Діаграма прецедентів, або Use-case діаграма представляє з себе спосіб опису головних функцій розробляємої системи, у нашому випадку захищеної системи розумного будинку. Першочерговою задачею при проектуванні системи є розробка такої діаграми, з метою описати й окреслити основні функції й можливості користувача. Тобто, вона розглядає модулі та компоненти нашого додатку у вигляді функцій та задач, котрі програма має виконувати. Use-case діаграма складається з таких елементів як актори, що являють собою користувача та можливих дій, що можуть виконувати актори, котрі часто називають прецедентами [17]. Такі дії є окремими випадками використання функцій програми. Між акторами й можливими опціями існує чітко визначений зв’язок, котрий можна спостерігати на діаграмі.

У ході виконання проектування системи розумного будинку в якості веб-додатку, було розроблено діаграму прецедентів, котру зображено на рисунку 2.1.

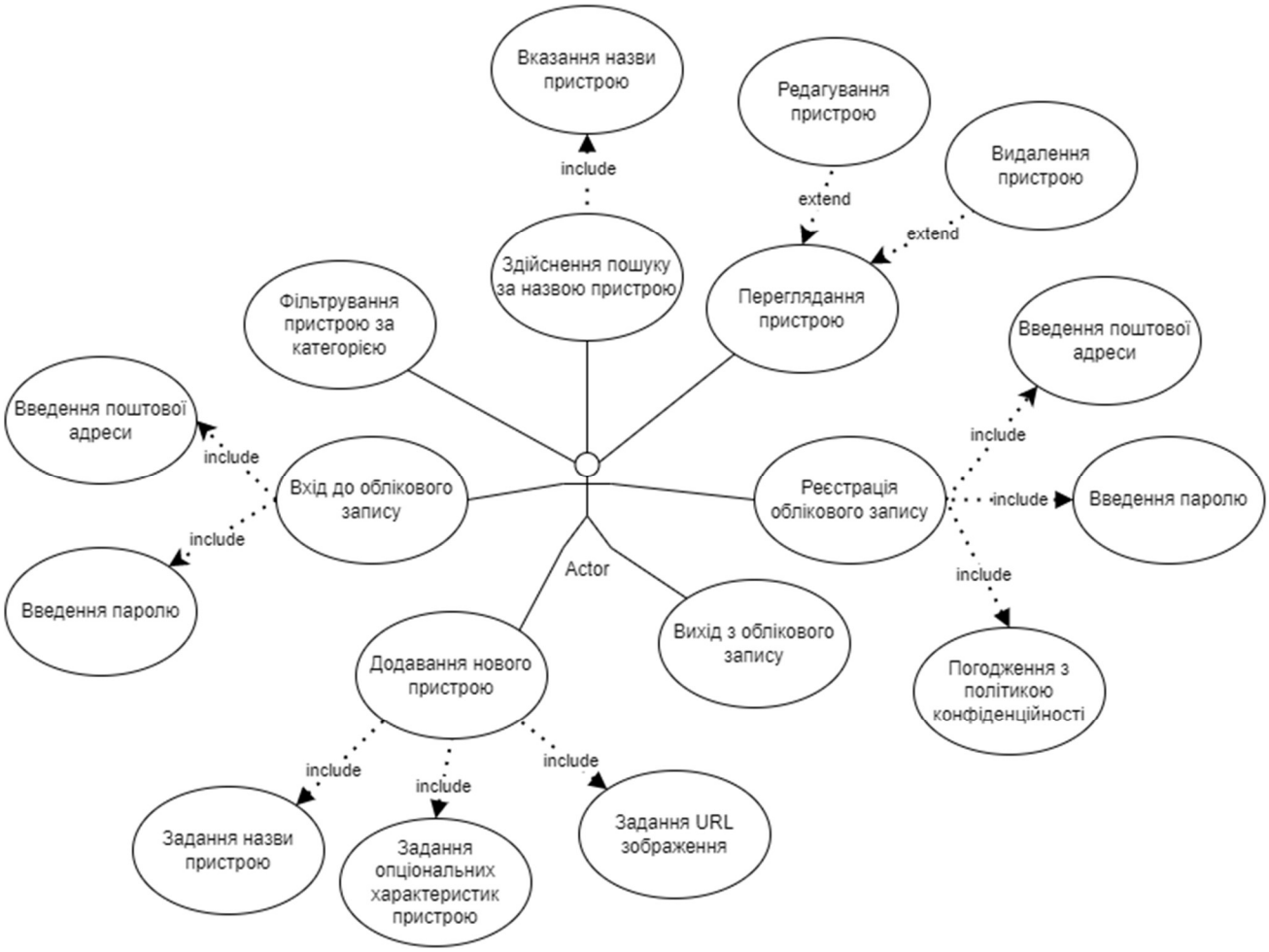

Рисунок 2.1 – Реалізація use-case діаграми

Іншою реалізованою в ході проектування системи розумного будинку стала діаграма послідовності. Така структура як діаграма послідовності, як і Use-case діаграма являє собою UML-діаграму, яка має собі за мету описати й наглядно представити можливості системи. Вона описує поведінку програми, опираючись на дії користувача застосунку у тому порядку, у якому ці дії відбуваються. На діаграми можна спостерігати поведінку програми та її складових частин у ході роботи системи. Такий сценарій роботи програми описується за допомогою елементів діаграми послідовності, серед яких є об’єкти системи й деякі повідомлення, за допомогою яких ці об’єкти комунікують між собою [18].

У ході виконання проектування системи розумного будинку в якості веб-додатку, було розроблено діаграму послідовності створення нового облікового запису користувача, котру зображено на рисунку 2.2.

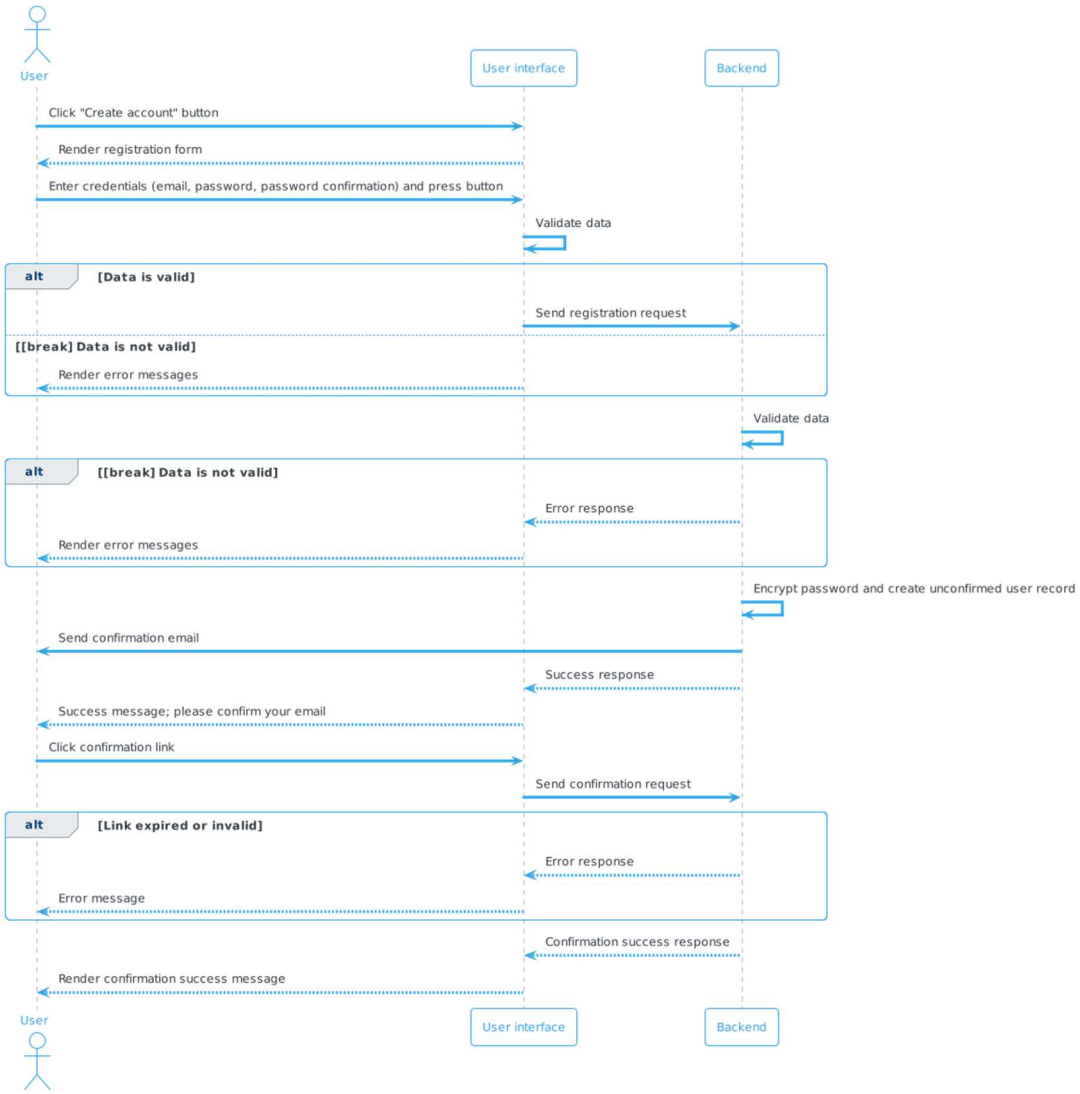

Рисунок 2.2 – Діаграма послідовності створення нового облікового запису юзера

Було розроблено діаграму послідовності автентифікації юзера, що зображено на рисунку 2.3.

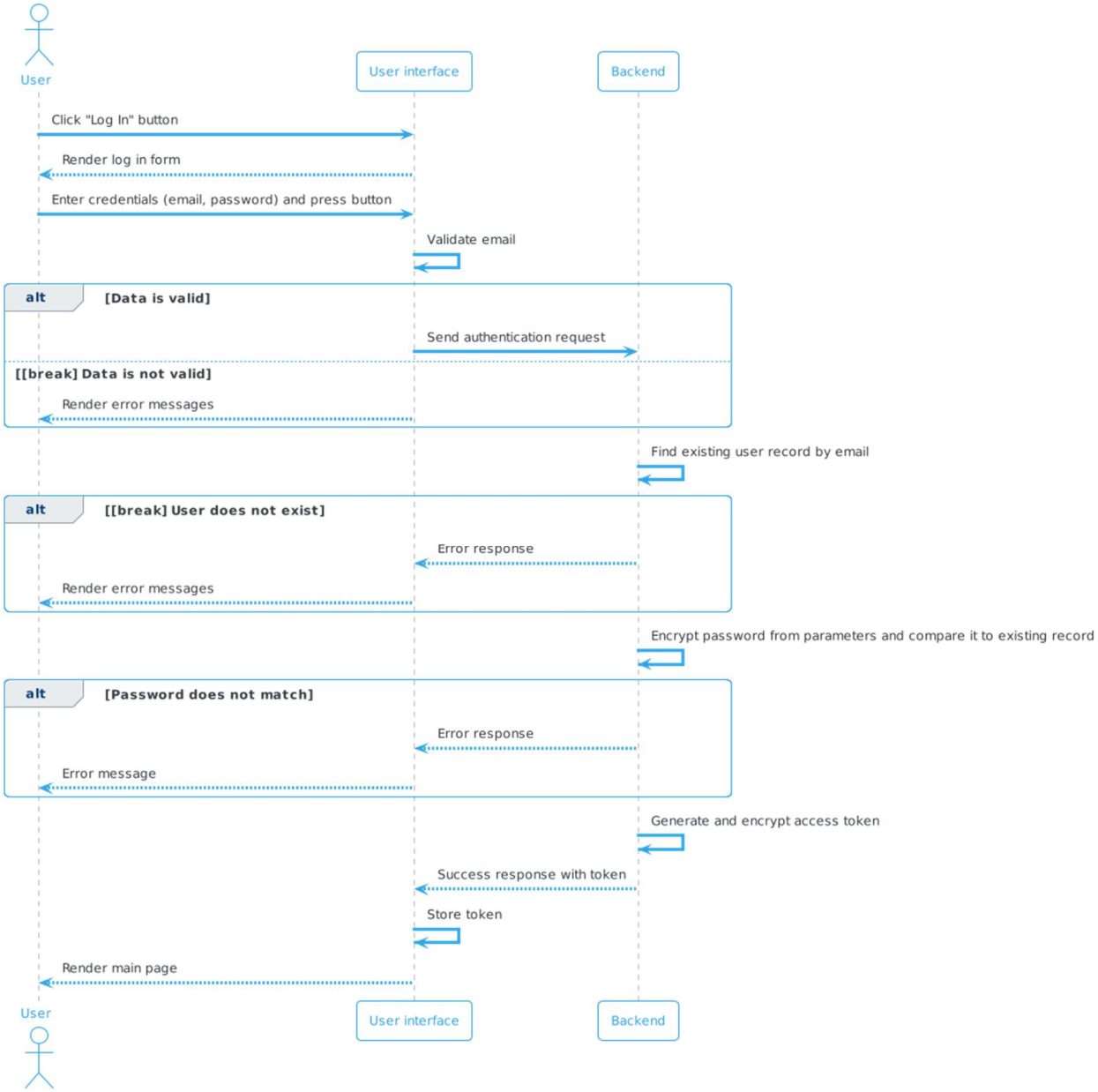

Рисунок 2.3 – Діаграма послідовності автентифікації юзера

Також в ході проектування системи розумного будинку була реалізована така структура як модель сутностей. Вона представлена у вигляді сутностей-таблиць з відповідними полями й зв’язків між такими таблицями [19]. Таким чином, розуміючи всі зв’язки між структурами ми можемо бачити перед собою всі обмеження по роботі з БД та даними. Даним способом ми можемо наглядно представити логічну структуру нашої БД, краще зрозуміти принципи її роботи та представлення даних у нашій розроблюваній системі.

У ході виконання проектування системи розумного будинку в якості веб-додатку, було розроблено модель сутностей та зв’язків між ними, або іншими словами ER-діаграма, котру зображено на рисунку 2.4.

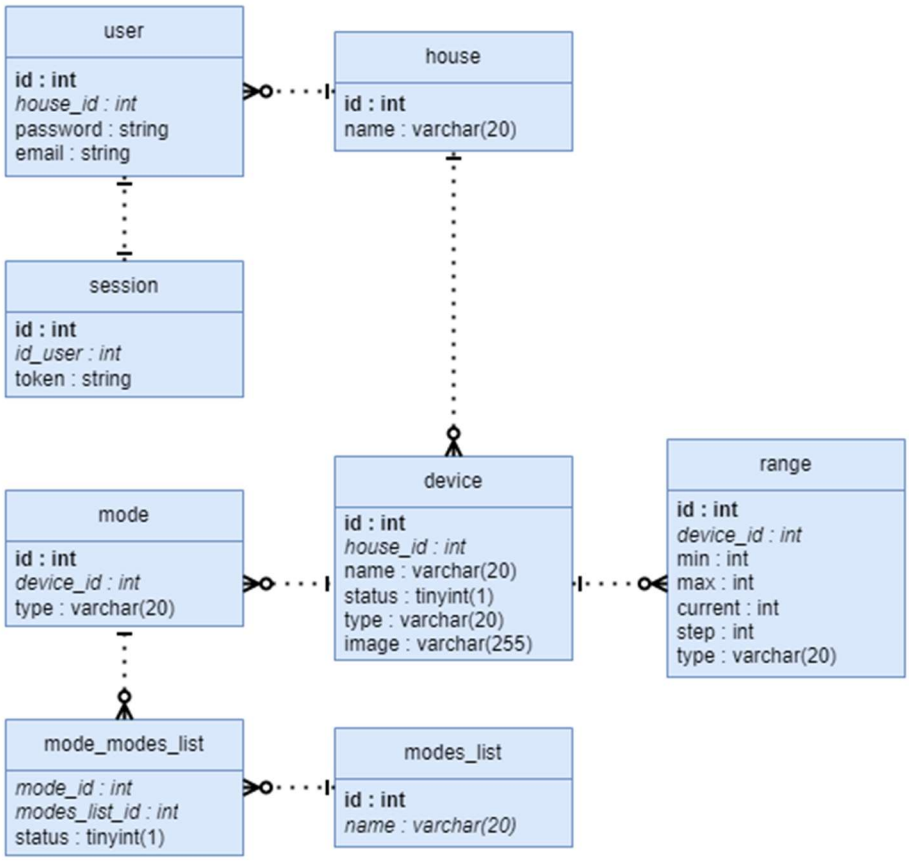

Рисунок 2.4 – Реалізована модель сутностей та зв’язків між ними

2.6 Вибір технологій для розробки веб-застосунку

Розроблювана система розумного будинку представляє собою захищений веб-додаток на базі клієнт-серверної архітектурної моделі, котрий має забезпечувати користувачам чудовий досвід використання програми на різних пристроях та ОС і в той же час використовувати відповідні принципе забезпечення безпеки роботи системи в цілому. В якості основи роботи клієнт-серверної системи було реалізовано REST підхід. Даний підхід забезпечує комунікацію між клієнтською й серверною частинами програми завдяки обміну повідомленнями, котрі називають запитами. Такі повідомлення реалізуються завдяки передачи даних у гнучкому й в той же час зручному форматі передачі даних JSON.

Обрані для розробки системи технології мають дозволяти системі гарно масштабуватися, щоб забезпечити можливість використання застосунка одразу великою кількістю користувачів. Також, серверна частина має бути в належною мірою захищеною від доступу з зовні, не допускати можливу втрату даних через використання відповідних вразливостей системи й працювати в умовах асинхронності, щоб оптимізувати навантаження на систему.

В даному розділі розглядаються технології, котрі будуть слугувати фундаментом розроблюваної системи, будуть відповідати поставленим нами вимогам до захисту й забезпечувати гарним прикладом використання сучасних підходів до реалізації безпечних веб-систем.

2.6.1 Технології розробки клієнту

В якості основної технології для розробки клієнтської частини додатку було обрано технологію ReactJS. Бібліотека ReactJS або, як її часто називають, React завдяки своїй універсальності та гнучкості являє собою найпоширеніше рішення для написання клієнтської частини веб-застосунків. Технологія працює на базі мови програмування JavaScript й слугує способом покращення написання веб-додатків найбільш ефективним зі способів. Використовується для створення інтерактивних користувацьких інтерфейсів на веб-сторінці, завдяки JSX синтаксису. JSX застосовують в умовах потреби в явному представленні користувацького UI (User Interface), завдяки структурному використанню коду JS та синтаксису HTML в одному файлі методом створення DOM (Document Object Model) дерев.

React виконує свою роботу з дотриманням MVC, котрий являє собою деякий архітектурний шаблон за яким маршрутизація програми виконується саме на клієнтській стороні додатку, що забезпечую динамічне оновлення веб-сторінки, тобто оновлення тільки потрібних для зміни частин.

Також, серед особливостей UI бібліотеки React можна виділити застосування функції Virtual DOM [20]. Вона являє собою технологію, котра забезпечує більш ефективну роботу, за звичайний DOM. Це досягається завдяки

тому, що в програмі при оновлені статусів компонентів, більш ефективна функція React оновлює лише виправлені елементи системи, замість того щоб оновлювати одразу їх всі в звичайному DOM. Таким чином, перезавантаження відбувається одразу ж як тільки відбулося оновлення якого-небудь елементу, що дозволяє якісно й швидко відображати актуальний користувацький UI.

Серед сучасних методів оптимізації, що пропонує ReactJS для забезпечення кращої безпеки веб-застосунку можна виділити наступні [21]:

- Узвичаєний захист від міжсайтового скриптингу, тобто XSS за допомогою використання технології прив’язки даних.
- Відстеження сумнівних URL посиланнь та контроль впровадження стороннього шкідливого програмного коду, котре може здійснюватися через URL.
- При розміщенні HTML коду, його можна розмістити відразу у DOM вузли, але така операція потребує ретельної перевірки та попереднього знезараження.
- При розміщенні коду до вузлів об’єктної моделі документа, тобто DOM, існує змога ухилятися та не допускати прямого відкритого доступу, що забезпечить відповідний захист.
- Використання корисних функцій React для серверного відображення даних на клієнті. Прив’язка даних надасть змогу екранувати дані при їх візуалізації на клієнтській частині застосунку у автоматичному режимі.
- Регулярна перевірка стану вразливостей та їх індекс, використовуючи інструмент OWASP Dependency-Check. Дана технологія перевірки вразливостей надає змогу виявляти відомі вразливості, котрі розташовуються у залежностях проекту, перш ніж додавати їх до розроблюваного проекту.
- Змогу запобігати вразливостям, котрі працюють за принципом впровадження даних JSON та ухиляйтеся від відповідних атак на систему.

Переважно такі дані відображаються на боці сервера та передаються разом зі сторінками React у форматі обміну даними JSON.

- Своєчасне оновленнями бібліотеки, задля використання найбільш безпечних її версій та підтримування стабільної версії React позбавляє багатьох вразливостей, котрі були слабким місцем у минулих її версіях.
- Використання лінтерів, запобігає неякісному користуванню інструментами забезпечення безпеки, що вбудовані напряму у React.

2.6.2 Технології розробки серверу

У якості основної технології розробки серверної частини додатку було обрано NodeJS, що являється середовищем виконання мови програмування JavaScript, з використанням HTML та CSS. Технологія NodeJS дозволяє нам реалізовувати системи на основі JSON API, використовуючи JavaScript і для розробки клієнта, і для розробки серверної частини застосунку та спрощуючи розробку захищеної системи завдяки великій кількості доступних модулів, що надають розробникам пакети JSON та NPM [22].

NodeJS представляє собою однопоточне рішення, що дозволяє обробляти підвищену кількість HTTP запитів. Забезпечуючи відсутність буферизації даних та можливість працювати з БД та її даними, технологія NodeJS працює в умовах сповіщення про події, що першочергово дозволяє ніколи не чекати доки програмний інтерфейс серверу поверне дані, а відразу переходить до наступного виклику API.

Створені завдяки NodeJS системи є кросплатформенними, що дозволяє їм працювати на таких платформах, як: Windows, Linux, MacOS, мобільних пристроях тощо.

Задля полегшення роботи з NodeJS та забезпечення додаткового захисту системи використовувався фреймворк Express, котрий призначається для реалізації кросплатформенних програмних інтерфейсів на сервері застосунку, завдяки використанню NPM [23]. Дана бібліотека дозволяє системі легше співпрацювати з БД, забезпечуючи більш захищену систему сховища даних.

Використовуючи Express можна реалізувати такий захищений роутер, котрий буде працювати з використанням HTTP методів та шляхів для звернення до API. Тобто, у тому випадку, коли на сервер надсилається запит з клієнта, то запит надсилається саме по заданому маршруту, де його буде опрацьовано й далі на клієнт повернеться HTTP відповідь. Така відповідь представляє собою дані у форматі JSON та з HTTP статусом відповіді.

2.6.3 Технології розробки бази даних веб-додатку

БД являють собою деякі структури даних, котрі мають зберігатися для роботи системи в цифровому вигляді й мати визначені правила по взаємодії з ними. Керування такою структурою відбувається завдяки ПЗ, що називають СКБД. Завдяки СКБД можна здійснювати операції з даними, створювати запити до БД, використовуючи відповідний програмний інтерфейс.

В якості технології розробки спроектованої раніше системи розумного будинку, використовується реляційна БД MySQL, котра працює на базі мови структурованих запитів SQL та представляє собою деяке цифрове сховище даних програми. Дані в MySQL зберігаються деякій кількості таблиць, кожна з яких складається з стовпчиків та рядків з даними, що втілює у життя ідею збереження великих масивів користувацьких даних [24].

СКБД реалізується на базі якостей ACID, серед яких є: атомарність, узгодженість, ізольованість й довговічність. Захищеність MySQL гарантується гарно пропрацьованою системою паролів.

Задля спрощення роботи з системою БД було використано сучасну технологію Sequelize. Вона являє собою ORM (Object-Relational Mapping) для роботи з реляційними СКБД, на базі мови запитів SQL. Вона є не єдиною в своєму роді й не найбільш використовуваною, але її можна назвати перевіреним рішенням для розробки веб-систем. Використовуючи Sequelize технологію ми будемо обробляти записи БД, завдяки роботі з даними у вигляді об’єктів, мати доступ до методів міграції даних та гнучкої системи зв’язків [25].

Серед механізмів захисту, що надають MySQL та Sequelize можна виділити наступні:

- Використання привілегій користувачів задля контролю доступу до сховища даних, як до всіх сховищ даних на серверів, так і на визначених БД на базі MySQL;
- Створення юзерів та ролей з урахуванням заздалегідь їх параметрів безпеки;
- Можливість використання автентифікації беручи за основу ОС дозволяє уникнути неавторизованих юзерів, що забезпечує захист системи від небажаного доступу;
- Розширення для проведення аудиту сховища даних, задля забезпечення додаткового захисту даних, що зберігаються у хмарі MySQL;
- Використовуючи різні методи утаювання даних та їх подальше шифрування новітніми ефективними типами шифрування й використання цифрових підписів;
- Використання системи надійних паролів, що забезпечується обов’язковими вимогами до їх створення;
- Можливість забезпечення захисту від атак на рівні мережі, за допомогою брандмауера та сервісу знаходження проникнень;
- Можливість використання TLS й SSL шифрування для запобігання під час комунікації між сервером й клієнтом отримання доступу до пакетів даних;
- Можливість проведення перевірки даних під час проведення запитів до БД;
- Sequelize запобігає вразливостям, котрі пов’язані з використанням SQL ін’єкцій.

2.7 Опис розроблених компонентів

У ході реалізації веб-додатку на сервері було реалізовано програмний інтерфейс. Далі наведені шляхи отримання доступу до API, тобто до функціоналу серверу:

- api/homes/ – створити новий будинок, метод CREATE;
- api/homes/ – отримати всі будинки, метод GET;
- api/homes/:homeid – отримати будинок, метод GET;
- api/user/register/ – створити новий обліковий запис юзера, метод POST;
- api/homes/:homeid – видалити будинок, метод DETELE;
- api/homes/:homeid – оновити будинок, метод UPDATE;
- api/user/login/ – увійти до облікового запису юзера, метод POST;
- api/homes/:homeid/devices/ – створити новий пристрій, метод CREATE;
- api/homes/:homeid/devices/:id?page=[]&perPage=[]&type=[]&subname=[] – отримати пристрої, метод GET;
- api/user/logout/ – вийти з облікового запису юзера, метод DELETE;
- api/homes/:homeid/devices/:id – отримати пристрій, метод GET;
- api/homes/:homeid/devices/:id – видалити пристрій, метод DELETE;
- api/homes/:homeid/devices/:id – оновити пристрій, метод UPDATE;

У «Лістингу Б.1», «Лістингу Б.2» та «Лістингу Б.3» наведена реалізація маршрутизації на сервері веб-застосунку.

Згідно з реалізованою нами схемою БД розроблюваного веб-додатку було створено декілька схем БД, серед яких є:

- houseSchema;
- userSchema;
- deviceSchema;
- modeSchema;
- modeListSchema;
- sessionSchema;
- rangeSchema.

Реалізація розроблених схем наведена у «Лістингу Б.4».

2.8 Використані механізми захисту системи

В ході створення захищеної системи розумного будинку було реалізовано деякі методи захисту веб-застосунку, серед яких можна виділити наступні:

- При користувацькому вводі даних у поля вводу, на клієнті й на сервері відбуваються перевірки, фільтрації й валідації всіх видів вхідних даних. Також застосовано санітайзинг спеціальних символів, що не дозволяє проводити ін’єкції коду й запобігає XSS. Валідацію електронної пошти й паролю та санітайзинг наведено у «Лістингу Б.5»;
- Завдяки аудиту менеджеру пакетів було забезпечено використання останніх версій бібліотек та механізмів захисту;
- Для перевірки реалізованої системи її було завантажено на безкоштовний хостинг, де було застосовано засіб зв’язку з шифруванням HTTPS, завдяки використанню SSL сертифікату. До конфігурації серверу було додано файл сертифікату та файл ключа та відкрито порт 443, котрий дозволяє встановлювати захищений зв’язок. Також застосування SSL сертифікату дозволяє захистити систему від SQL вразливостей;
- Використано механізм обмеження клієнтів, що можуть робити запити на сервер для того, щоб запобігти запитам з усіх джерел, що не внесені до «білого списку», включаючи шкідливі що можуть використовуватися зловмисниками. Реалізовано за допомогою бібліотеки cors. Використання файлу оточення наведено у «Лістингу Б.6»;
- Для забезпечення захисту даних й системи було використано бібліотеку хешування паролів bcrypt, за допомогою якої було згенеровано сіль та з її допомогою захешували пароль, який ми потім зберігаємо у БД в хешованому вигляді. Кожного разу, коли користувач здійснює спробу входу до свого облікового запису, його введений пароль хешується й отриманий хеш порівнюється з тим, котрий зберігається у БД. Хешування паролю наведено у «Лістингу Б.7»;

- Завдяки реалізації системи автентифікації користувачів на основі токенів з використанням інструменту JWT (JSON Web Token), система має змогу підтвердити особу користувача. Токен зберігається на клієнті, що робить можливою авторизацію без повторного використання його логіну та паролю. Кожен токен містить цифровий підпис, що дозволяє нам підтвердити його дійсність та запобігти непередбаченому доступу до системи. Генерацію JWT токена наведено у «Лістингу Б.8»;
- Завдяки використанню модуля Helmet було встановлено різні HTTP заголовки, котрі забезпечують захист від XSS атак й клікджекінгу;
- В системі реалізовано алгоритм авторизації користувачів до ресурсів (пристроїв). Коли відбувається запит на API з ціллю редагування, видалення або ж зчитування даних про пристрій, то система перевіряє що користувач дійсно є власником даного пристрою. Таку логіку й функціонал авторизації дозволяє нам реалізувати інструмент pundit. Механізм авторизації наведено у «Лістингу Б.9»;
- Для запобігання атакам типу «грубої сили», реалізовано блокування можливості багаторазових спроб входу до облікового запису користувача. У разі введення неправильного паролю більше трьох разів, користувачу на електронну адресу прийде лист, інформуючий його про проблеми із входом. Механізм захисту від атак «грубої сили» наведено у «Лістингу Б.10»;
- При створенні нового облікового запису користувач повинен задати складний пароль, вимоги до якого вказано на клієнті. Валідація паролю при створенні акаунту значно ускладнює можливі спроби зламу облікового запису методами підбору паролю. Валідацію паролю на клієнті наведено у «Лістингу Б.11»;
- Для реалізації БД в умовах забезпечення додаткового захисту було використано ORM Sequelize, що дозволяє запобігти SQL ін’єкціям й досягти додаткової безпеки системи;

- Забезпечено використання безпечних параметрів для SQL запитів до БД. Завдяки відношенню до вхідних даних як до строки, а не частини запиту ми можемо запобігати SQL ін’єкціям. Використання безпечних параметрів для SQL запитів наведено у «Лістингу Б.12»;
- У програмі було вимкнено функції автозаповнення даних, що дозволяє запобігти розкриттю конфіденційних даних;
- Застосування логуванню всіх запитів на сервер дозволяє забезпечити можливість перегляду помилок, підозрілих запитів й спроб витоку даних. Логування етапів виконання системи наведено у «Лістингу Б.13»;
- Для уникнення вразливостей XXE у системі використовується формат передачі даних JSON, замість XML, що запобігає серіалізації конфіденційних користувацьких даних;
- Використано інструмент dotenv, котрий дозволяє зберігати дані поза кодом в окремому файлі для того, щоб забезпечити безпеку чутливих конфіденційних даних. Використання файлу оточення наведено у «Лістингу Б.14»;
- Застосування лінтерів дозволяє здійснювати стеження за якістю коду за сучасними стандартами.

2.9 Логіка роботи розробленої системи

Коли юзер вперше заходить на головну сторінку веб-додатку розумного будинку, то він спершу має створити свій обліковий запис. Так, він опиняється на сторінці створення облікового запису, де йому пропонується ввести адресу поштової скриньки та створити пароль. Натиснувши кнопку реєстрації й у тому випадку, якщо введені юзером дані задовольняють нашу систему й всі валідації на клієнті у режимі реального часу виявилися пройденими.

В ході процесу валідації відбувається перевірка даних, котрі було введено. Якщо логін та пароль юзера відповідають нашим вимогам, то валідацію буде пройдено вдало, буде відправлено запит на сервер з адресою поштової скриньки

користувача й паролем. Далі пароль буде захешовано й до БД буде додано дані новоствореного облікового запису з отриманим хешем, замість паролю й на клієнт з серверу буде відправлене повідомлення про вдале виконання процесу створення нового облікового запису. Хешування паролів й зберігання в БД результату хешування замість паролів дозволяє не зберігати паролі користувачів у відкритому вигляді й забезпечити безпеку юзерів.

У випадку, якщо введена адреса поштової скриньки виявиться дублікатом, тобто такий обліковий запис все буде існувати в системі, на клієнт буде відправлено помилка створення облікового запису.

На рисунку 2.5 зображено реалізовану в системі сторінку реєстрації нового юзера.

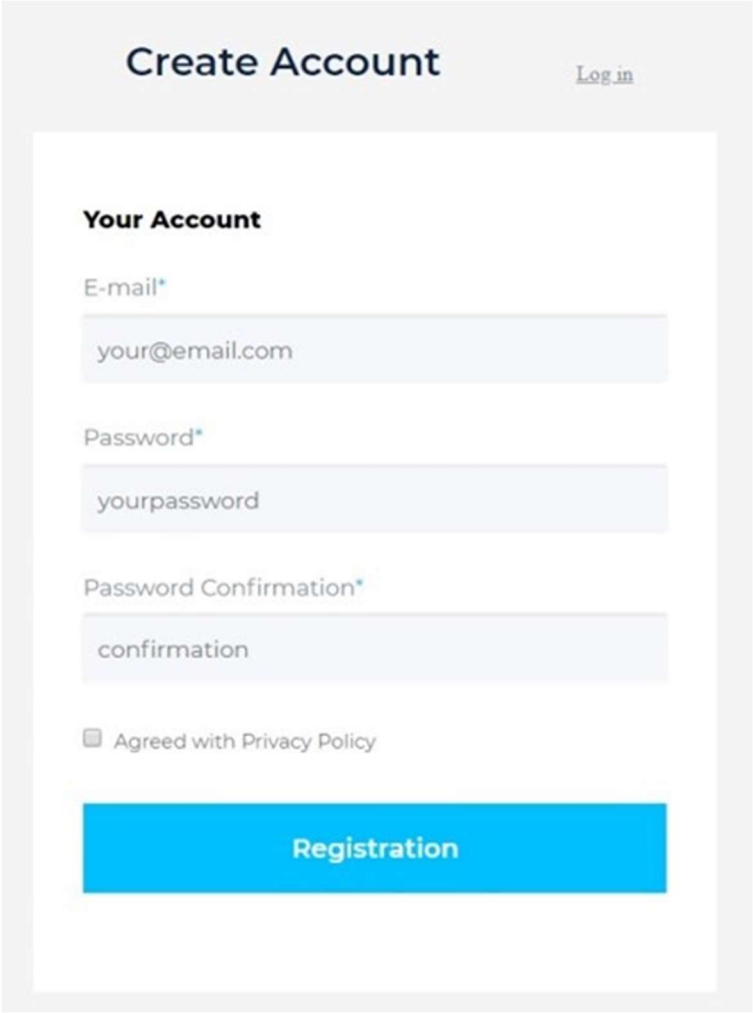

Рисунок 2.5 – Сторінка реєстрації нового юзера

Перейшовши до процесу входу до новоствореного облікового запису юзер має надати дані свого облікового запису до форми входу й ініціювати процес авторизації. В такому разі дані знову пройдуть процес валідації у реальному часі на клієнті й у разі вдалого проходження валідації буде виконано запит на сервер. Далі відбувається перевірка на існування такого облікового запису у БД, хешування введеного паролю й порівняння з хешем пароля, котрий вже зберігається у БД. Такий механізм авторизації користувача дозволяє уникнути багатьох проблем безпеки й вразливостей та слугує гарною практикою забезпечення захищеності системи. Процес авторизації реалізується завдяки ідеї хешування паролів. Це є необхідністю для забезпечення безпеки даних користувача. Сторінка входу до облікового запису юзера зображена на рисунку 2.6.

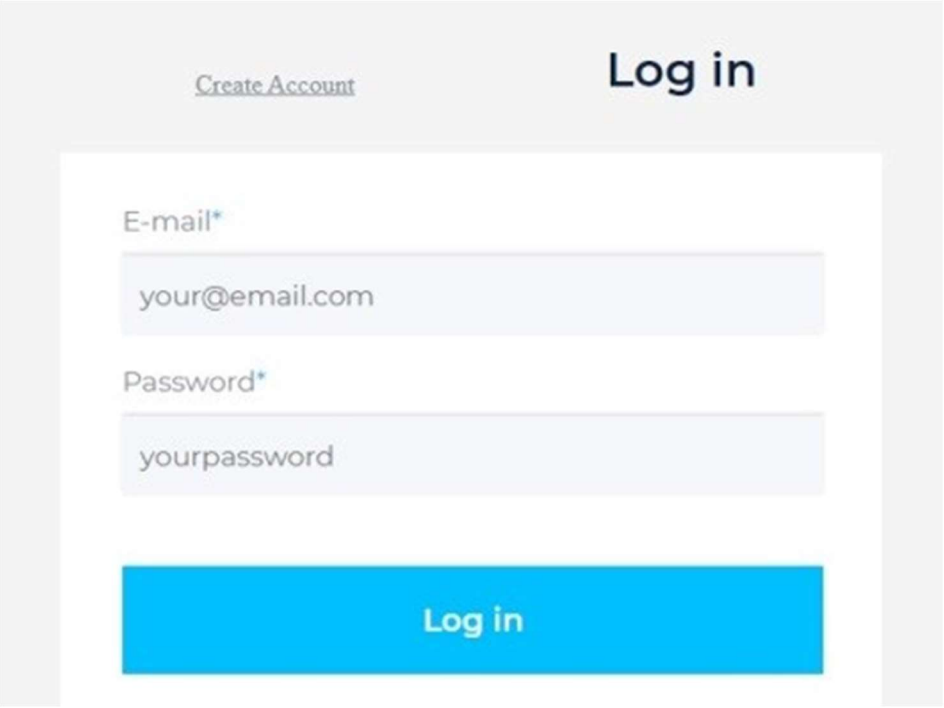

Рисунок 2.6 – Сторінка входу до облікового запису юзера

Зробивши вхід до свого облікового запису юзер опиняється на головній сторінці програми. В головному вікні програми юзер може додавати нові пристрої до системи розумного будинку, здійснювати сортування за категоріями пристроїв й здійснювати пошук конкретного пристрою за назвою. На даний момент існує дві категорії пристроїв: Піч та Робот.

На рисунку 2.7 представлено головну сторінку веб-додатку.

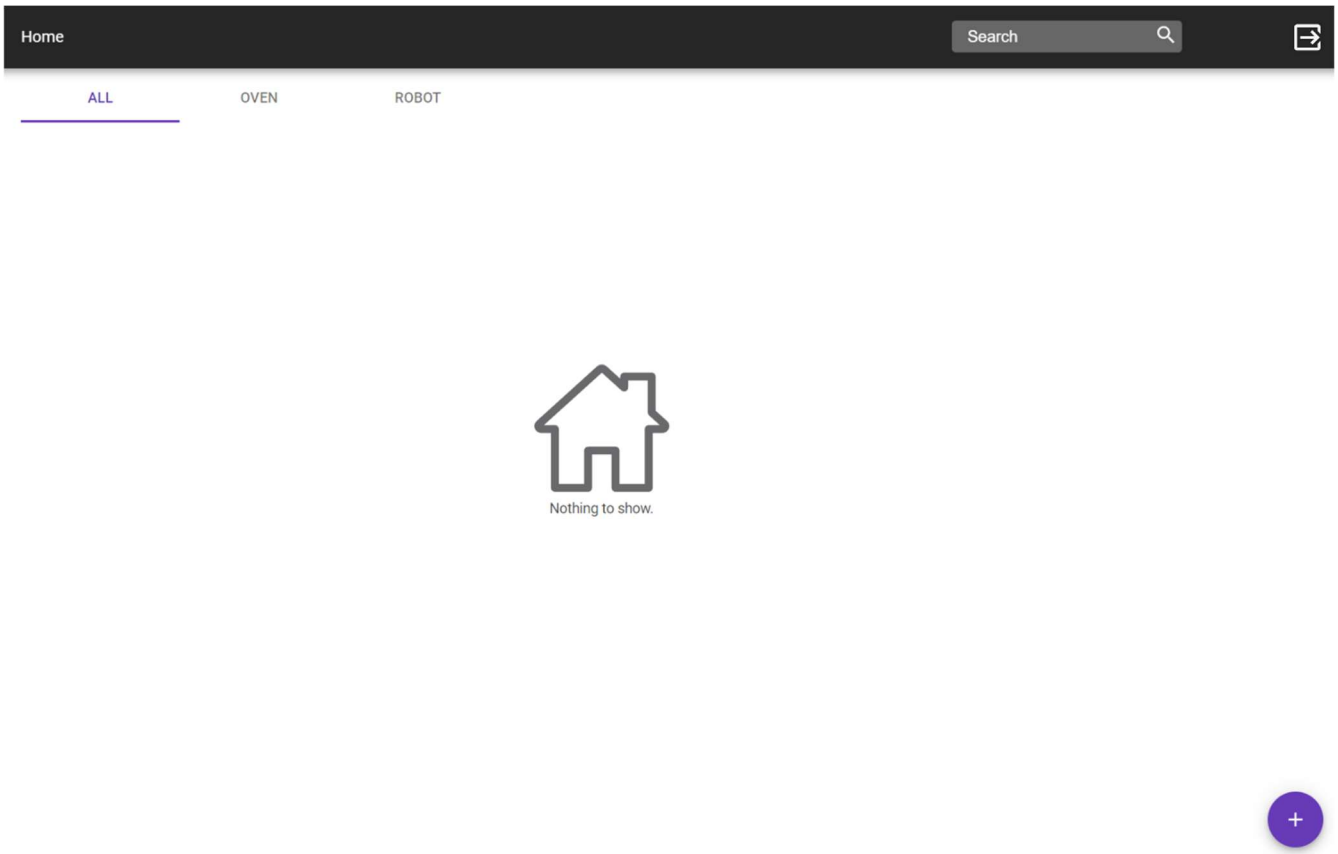

Рисунок 2.7 – Головне вікно програми розумного будинку

Натиснувши кнопку додавання нового пристрою юзеру відкривається модальне вікно з можливістю створення, використовуючи яке необхідно обрати його тип. Це зображено на рисунку 2.8.

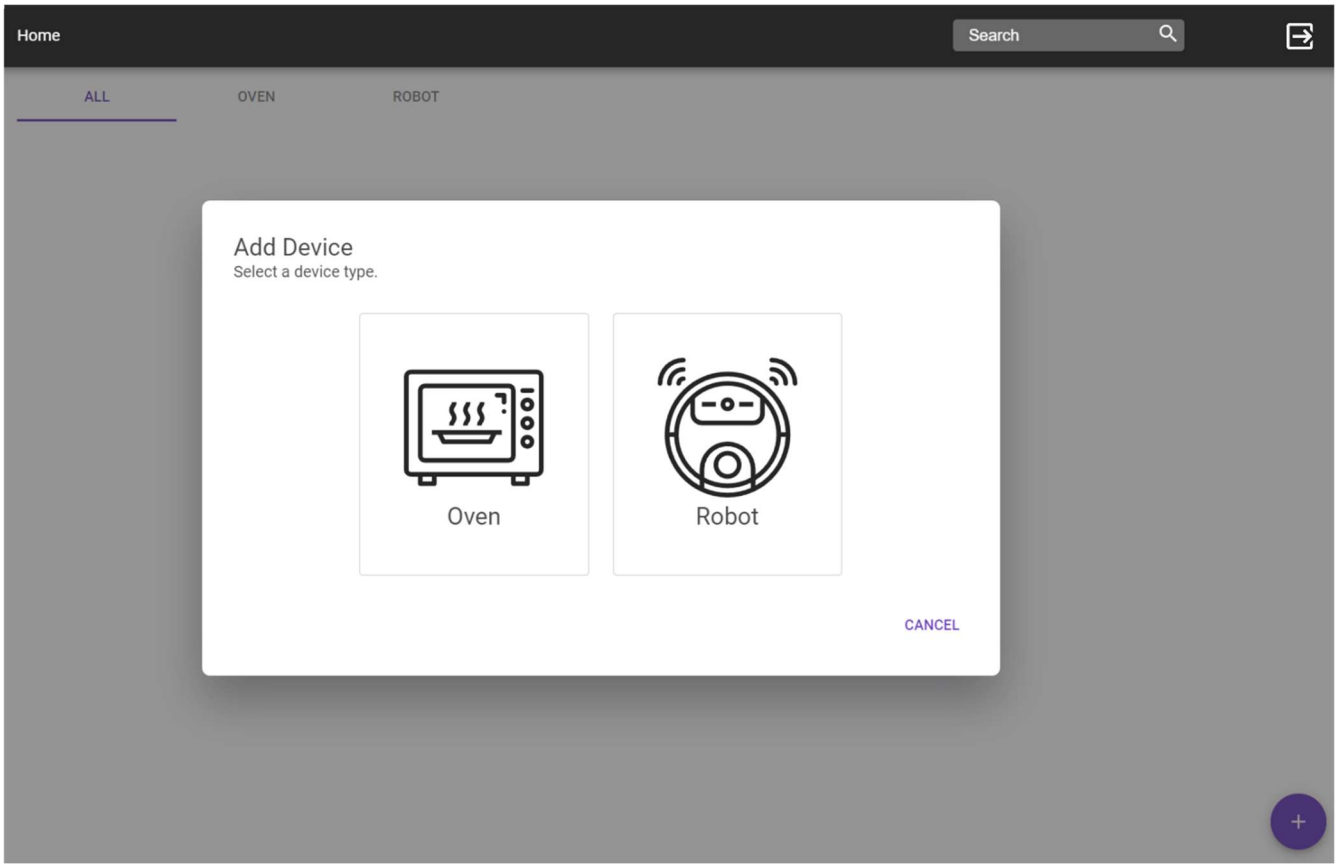

Рисунок 2.8 – Сторінка обирання категорії нового пристрою

Наприклад, обравши тип пристрою Піч, юзер бачить вікно створення пристрою категорії Піч, де йому запропоновано задати пристрою назву, URL-адресу, зображення пристрою, задати режими роботи пристрою та деякі властивості, котрі залежать безпосередньо від типу пристрою. В даному випадку, юзеру пропонується задати певні рамки показнику температури пристрою: мінімальну, максимальну, поточну й крок зміни температури. Це зображено на рисунку 2.9.

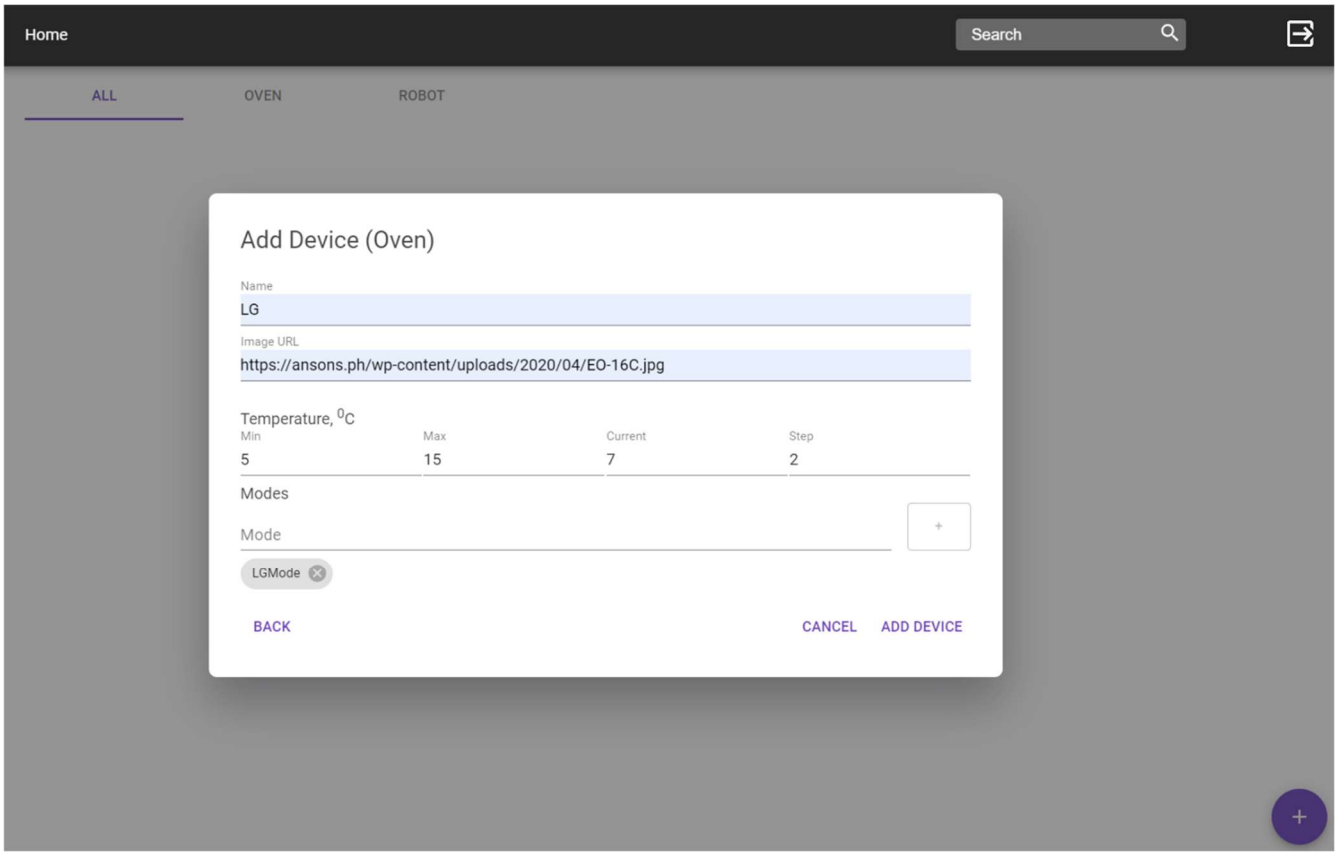

Рисунок 2.9 – Сторінка створення нового пристрою

Після натискання кнопки додавання нового пристрою, з клієнта на API відбудеться запит створення нового пристрою, котрий буде провалідовано на сервері. У випадку вдалого проходження валідації, буде створеного запис нового пристрою у БД з параметрами що було отримано з клієнта. Після додавання нового пристрою у БД на клієнт буде повернено відповідь з кодом 201 й створеним пристроєм, котрий відображається на головній сторінці.

На рисунку 2.10 зображено створені пристрої, котрі ми бачимо на головній сторінці веб-додатку.

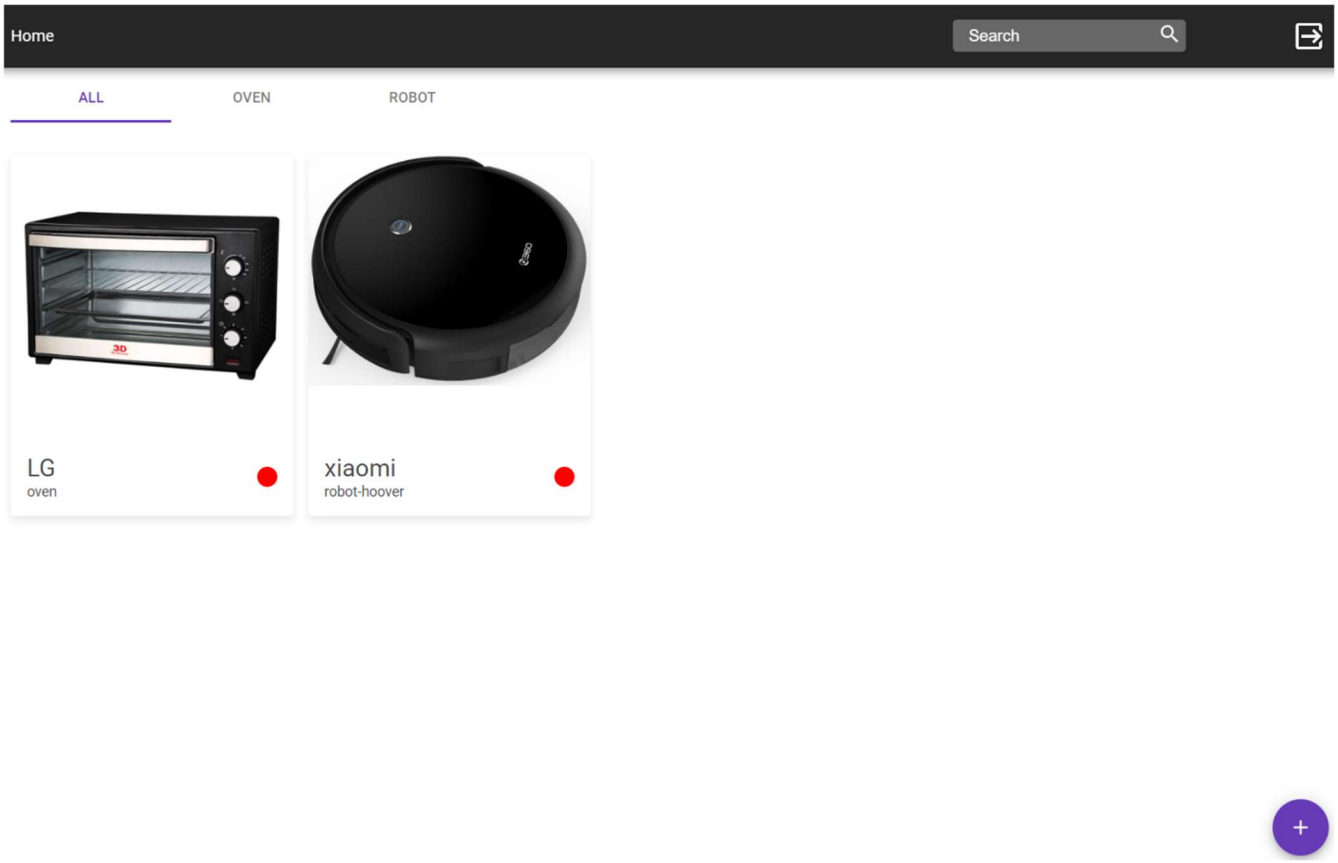

Рисунок 2.10 – Головна сторінка додатку з доданими пристроями

Обравши зображений на головній сторінці пристрій, користувачу відкривається вікно редагування пристрою. Там ми можемо керувати станом роботи пристрою (вмикати або вимикати), змінювати персональні властивості пристрою (в випадку пристрою категорії Піч ми можемо змінювати температуру роботи), обирати режим роботи з доступних нам режимів тощо. Також на сторінці редагування пристрою знаходяться кнопка видалення пристрою й кнопка збереження змін.

Коли юзер натискає кнопку видалення пристрою, ініціюючи процес видалення, то з клієнта API відправляється запит типу DELETE, запис пристрою видаляється з БД, на клієнт повертається повідомлення з кодом 200, що означає успіх й сторінка додатку оновлюється.

У випадку редагування пристрою й натисканні кнопки збереження змін, з клієнта на API відправляється запит PUT. Сервер шукає існуючий запис у БД з вказаним ідентифікатором й у разі знаходження відбувається валідація

параметрів, запис оновлюється й на клієнт повертається відповідь з кодом 200, котра означає успіх.

Вікно редагування пристрою представлено на рисунку 2.11.

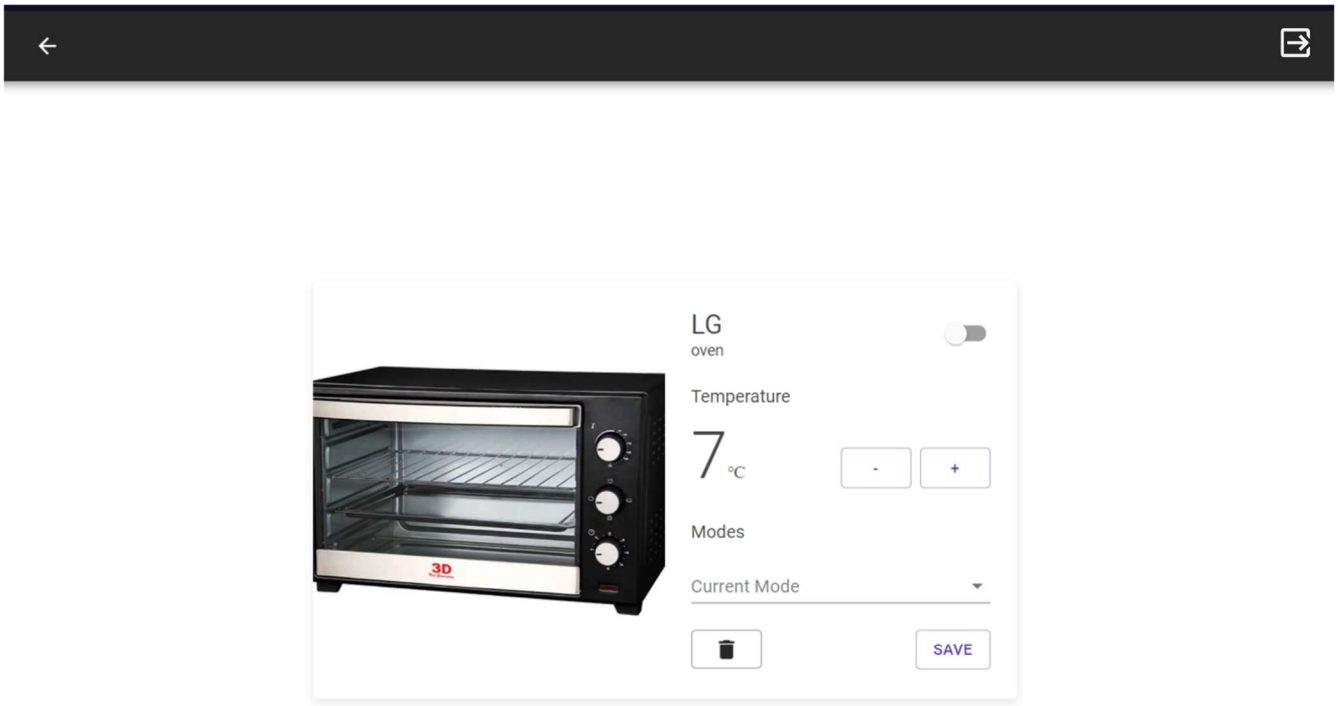

Рисунок 2.11 – Вікно редагування пристрою

На головній сторінці є поле для пошуку існуючого пристрою за ім’ям пристрою та сортування пристроїв за їх типом. З клієнта на API надсилається GET запит з параметрами пошуку запису у БД. Сервер впевнюється в тому, що отриманий запит з параметрами не буде виконано як SQL запит напряму. Після валідації здійснюється пошук запису в БД по заданому параметру й на клієнт повертається список, який може містить або не містити пристрою, в залежності від результатів пошуку. Повертається код 200, що значить проведення успішної операції.

Пошук пристрою за ім’ям показано на рисунку 2.12.

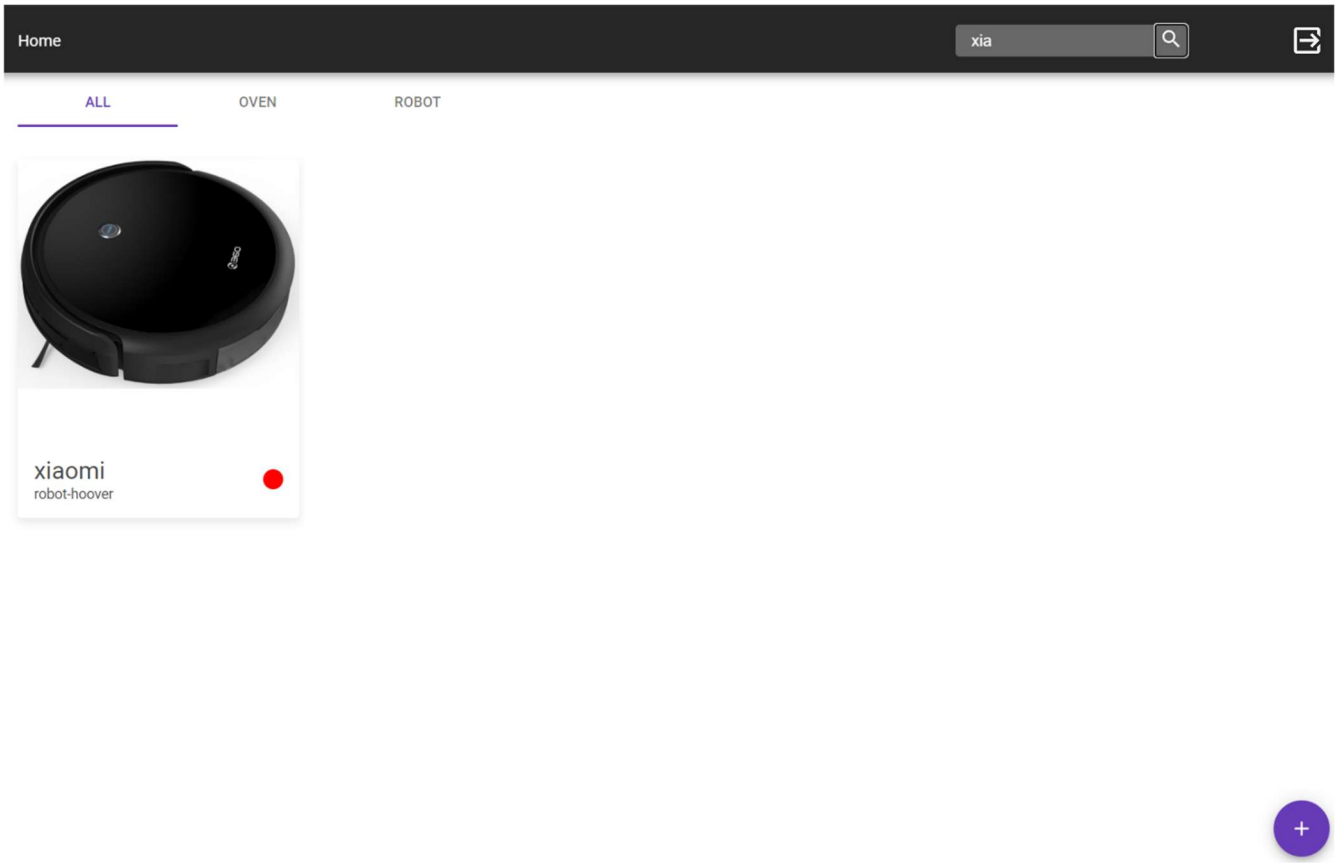

Рисунок 2.12 – Пошук пристрою за ім’ям

3 ТЕСТУВАННЯ ЗАСТОСУНКУ

У даному розділі приводяться різнотипні методи тестування системи, котрі допомагають зрозуміти чи правильно була розроблена система й побачити базові помилки, як функціональні, так і захисту. Серед деяких з проведених тестувань можна виділити наступні: мануальне тестування, юніт тестування, наскрізне тестування, яке ще називають end-to-end та тестування сканером коду.

3.1 Мануальне тестування системи

Тестування системи методом ручного, або мануального тестування являє собою такий спосіб перевірки якості розробленого продукту, використовуючи який розробник або тестувальник ПЗ перевіряє деякі можливості розробленої системи не використовуючи при цьому способи автоматизувати процес. Мануальне тестування являється гарним способом виявлення загальних помилок системи, що лежать на поверхні й в повній мірі всеціло використовується в командах та компаніях при розробці їх програмних рішень. В ході перевірки системи розробник ставить себе на місце користувача й виконує той набор дій, котрий мав би виконати юзер, таким чином перевіряючи відповідність розробленої системи раніше поставленим до ней вимогам [26].

Протестувавши систему мануальним тестуванням можна зробити висновок про необхідність використання методів автоматизованої перевірки якості системи. Оскільки перевірити розроблену систему на наявність недоліків є нашою головною метою, веб-додаток спершу було перевірено саме в мануальному режимі.

У випадку якщо юзер здійснює спробу вводу некоректних даних під час реєстрації свого облікового запису через форму реєстрації додатку, то він бачить відповідне інформаційне вікно з інформацією введення ним некоректних даних у недозволеному системою форматі. Це зображено на рисунку 3.1.

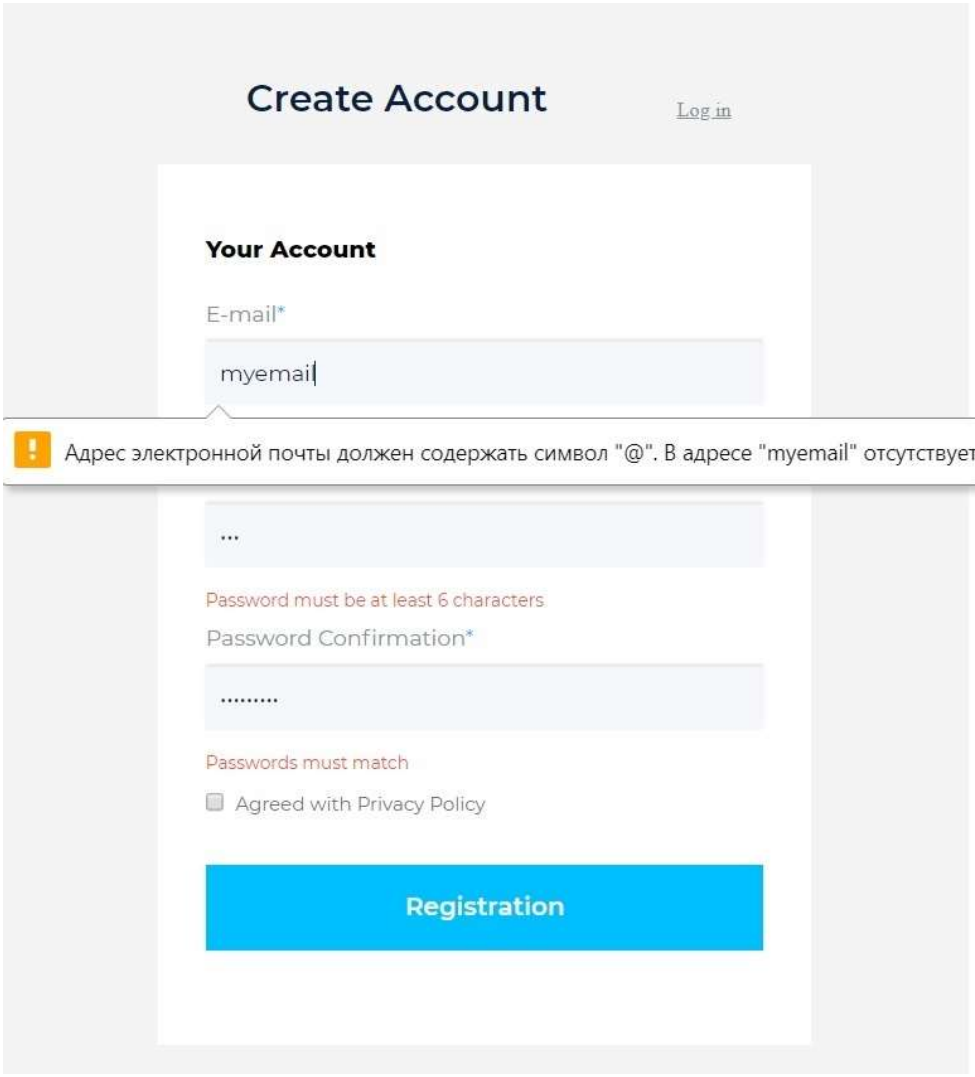

Рисунок 3.1 – Спроба введення даних при реєстрації нового облікового запису у недозволеному системою форматі

У випадку введення даних у недозволеному програмою форматі в процесі здійснення входу до вже існуючого облікового у програмному інтерфейсі застосунку юзеру робляться відповідні підказки, що дозволяють зрозуміти суть проблеми та її вирішити. Такий процес реалізується завдяки валідації даних на клієнті у режимі реального часу. З такою поведінкою системи можна ознайомитися на рисунку 3.2.

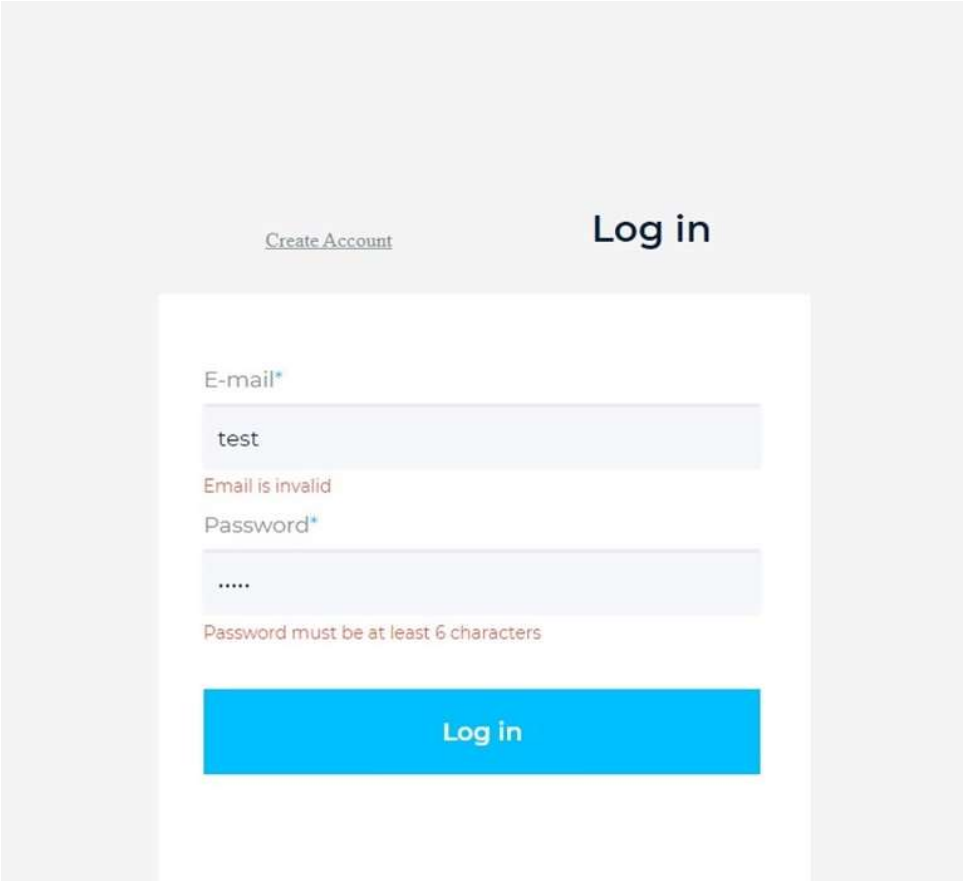

Рисунок 3.2 – Спроба введення даних при вході до облікового запису користувача у недозволеному системою форматі

На рисунку 3.3 представлено реакцію системи на введення свідомо некоректних даних. Якщо облікового запису з введеними юзером даними не існує у БД або якщо введений пароль відрізняється, юзер побачить повідомлення, котре проінформує його про проблему зі входом. В такому разі юзер зможе ввести інші дані облікового запису й спробувати здійснити вхід ще раз. Спробу введення некоректних даних зображено на рисунку 3.3.

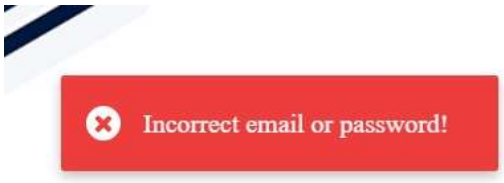

Рисунок 3.3 – Спроба ввести неіснуючі або некоректні дані існуючого облікового запису користувача

В разі спроби додавання нового пристрою до свого розумного будинку та введенні даних які не пройшли початкову валідацію на клієнті, юзер побачить

відповідні повідомлення. Виправивши помилки, юзер зможе додати новий пристрій. Помилки при додаванні нового пристрою можна спостерігати на рисунку 3.4.

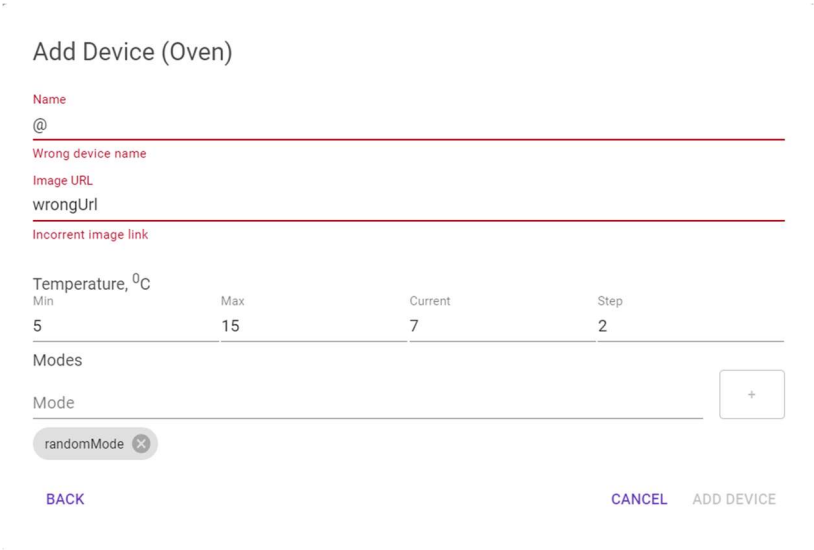

Рисунок 3.4 – Приклад помилки при додаванні нового пристрою

В ході проведення мануального тестування були використані декілька різних ОС та браузерів, що було необхідністю для точного розуміння працездатності розробленої системи розумний будинок у разних умовах використання застосунку.

Протестувавши розроблену систему методом мануального тестування не було помічено критичних недоліків системи, тому виконане тестування можна вважати вдалим й можна переходити то методів автоматизованого тестування, серед яких можна виділити наскрізне тестування, тобто end-to-end й тестування завдяки реалізованим юніт тестам.

3.2 Автоматизоване тестування системи

Автоматизоване тестування являє собою зручний метод проведення тестів з метою контролю якості розробленою системи й перевірки реалізованого функціоналу застосунку. Цей вид тестування є дуже гнучким та дозволяє перевірити відповідність реалізованого функціоналу системи до раніше визначених вимог, займаючи при цьому значно менший час, аніж ручне тестування й відповідно зменшуючи вірогідність пропустити який-небудь недолік системи [27].

Автоматизоване тестування може бути реалізованим за будь-якими потрібними тестувальнику сценаріями. Такі сценарії частіше за все включають в себе деяку кількість однотипних дій направлених на систему, котрі маніпулюють системою завдяки різним вхідним даним.

Багато видів автоматичного тестування притримаються принципів CI/CD (Continuous Integration/Continuous Delivery), що робить можливим запустити їх в той момент, коли вони потрібні, без гострої необхідності в їх редагуванні. Цей принцип дозволяє виконувати тестування системи велику кількість разів.

Протестувати систему автоматизованими видами тестування ми можемо як у вигляді тестів API чи на конкретні програмні компоненти, так і у вигляди тестів що направлені на програмний користувацький інтерфейс. Таким чином, перевіривши розроблене ПЗ ми маємо гарну можливість порівняти очікувану нами реакцію системи на тестування її функцій з тою реакцію, котру ми бачимо в ході виконання тестів.

В ході написання автоматизованих тестів, задля перевірки якості реалізованої системи розумного будинку, було реалізовано такі види тестування як: юніт тестування й наскрізне тестування end-to-end.

Юніт тестування перевіряє відокремлені складові частини реалізованої системи, таким чином дозволяючи розробнику переконатися у тому, що кожен модуль системи працює саме так, як задумувалося розробниками системи. Таким чином, провівши якісне юніт тестування застосунку, можна передчасно виявити баги й бути на сто відсотків впевненим у тому, перевірена системні модулі не мають дефектів. Методом перевірки кожної окремою маленької частинки системи, серед яких є різного роду модулі та функції [28]. Юніт тестування являє собою реалізацію SAST підходу до тестування якості розроблюваного ПЗ.

Попри те, що існує багато різних інструментів створення й виконання юніт тестування, використовуватися буде середовище тестування Jest, що являє собою фреймворк, направлений на простоту. Для роботи з Jest нам не потрібно виконувати додаткові додаткову відладку, щоб тестувати систему на базі ReactJS

та NodeJS. Працює в режимі виконання тестів у паралельному режимі, що дозволяє йому бути найбільш ефективним.

Програмна реалізація написаних юніт тестів показана у «Лістингу Б.15».

Результат виконання юніт тестів зображено на рисунку 3.5.

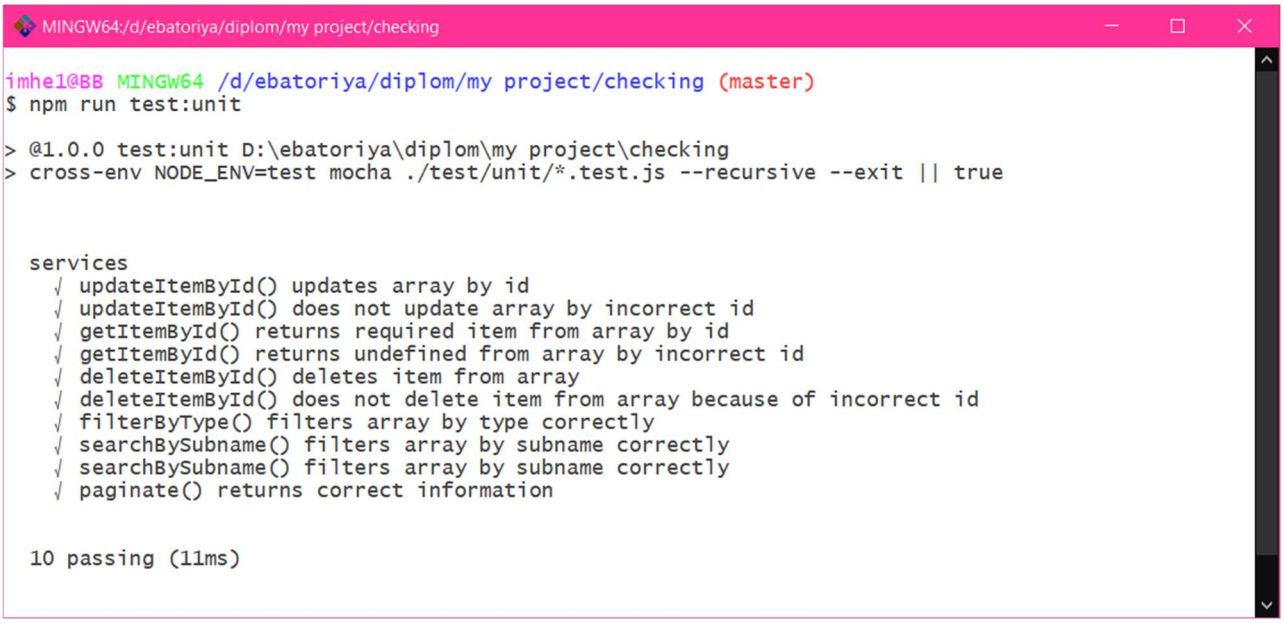

Рисунок 3.5 – Результат виконання юніт тестів

End-to-end або наскрізне тестування представляє собою відому методологію тестування, котра здійснює тестування системи в таких умовах, котрі схожі на реальний повний сценарій використання додатку юзером. Відповідно до своєї назви, наскрізне тестування по своїй суті перевіряє систему та її підсистеми на те чи працюють вони як задумано [29]. Порівняно з юніт тестуванням, end-to-end тестування здатне виходити за рамки виділених модулів системи, таким чином, тестуючи як додаток працює з БД мережею тощо.

В умовах інструментів реалізації серверу на базі інструментів NodeJS та Express ми будемо тестувати систему end-to-end тестуванням, використовуючи інструмент проведення наскрізного тестування Supertest, котрий створено задля тестування HTTP API системи.

Програмна реалізація написаних end-to-end тестів показана у «Лістингу Б.16».

Результат виконання end-to-end тестування приведено на рисунку 3.6.

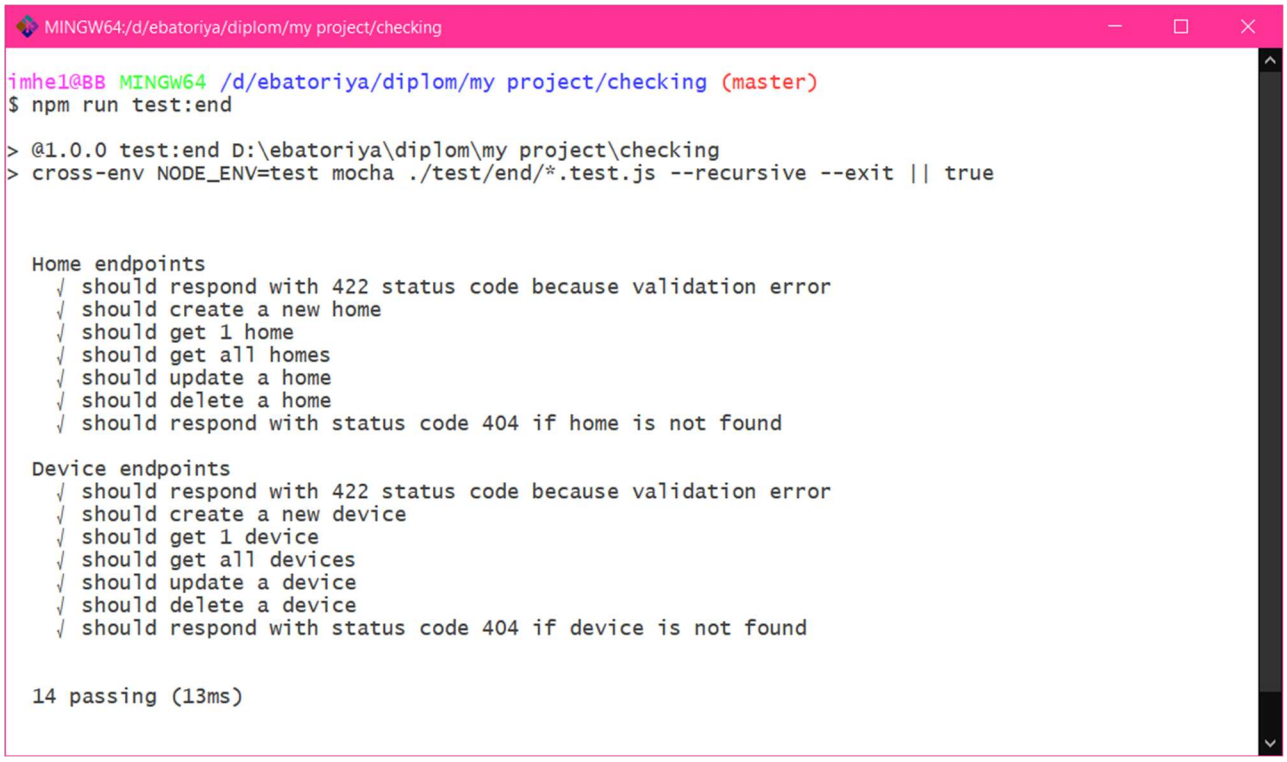

Рисунок 3.6 – Результат виконання end-to-end тестів

3.3 Тестування сканером коду

Для забезпечення якості розробленої системи й перевірки її на недоліки й вразливості, веб-застосунок було перевірено методом SAST завдяки сканеру програмного коду Snyk. Інструмент безпеки Snyk являє собою сучасний та ефективний інструмент контролю якості продуктів й виявлення вразливостей реалізованих систем, котрий забезпечує безпеку коду та складових частин системи. Здатний сканувати програмний код, що написано на мові програмування JavaScript [30].

На рисунку 3.7 зображено результати тестування клієнтської частини розробленої системи розумного будинку.

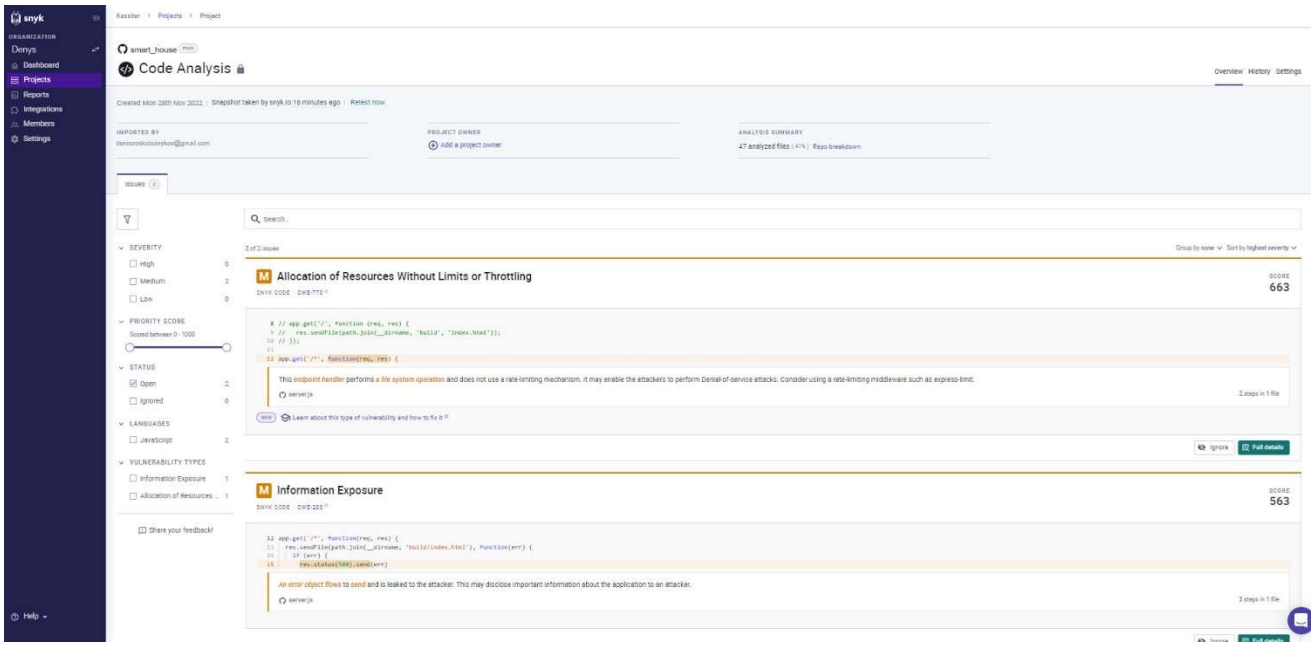

Рисунок 3.7 – Результат тестування клієнту сканером Snyk

На рисунку 3.8 представлено результати сканування серверу інструментом Snyk.

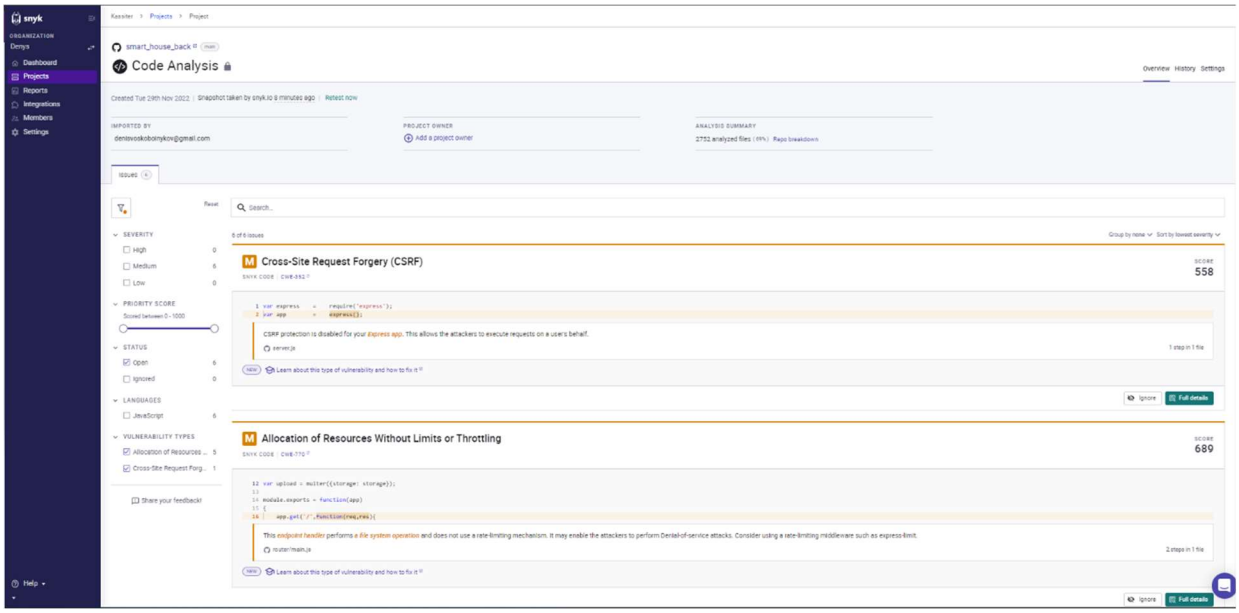

Рисунок 3.8 – Результат сканування серверної частини застосунку сканером Snyk

В ході проведення сканування інструментом Snyk було виявлено декілька некритичних помилок, котрі були виправлені.

На рисунку 3.9 представлено результат повторного сканування системи на предмет вразливостей й загроз.

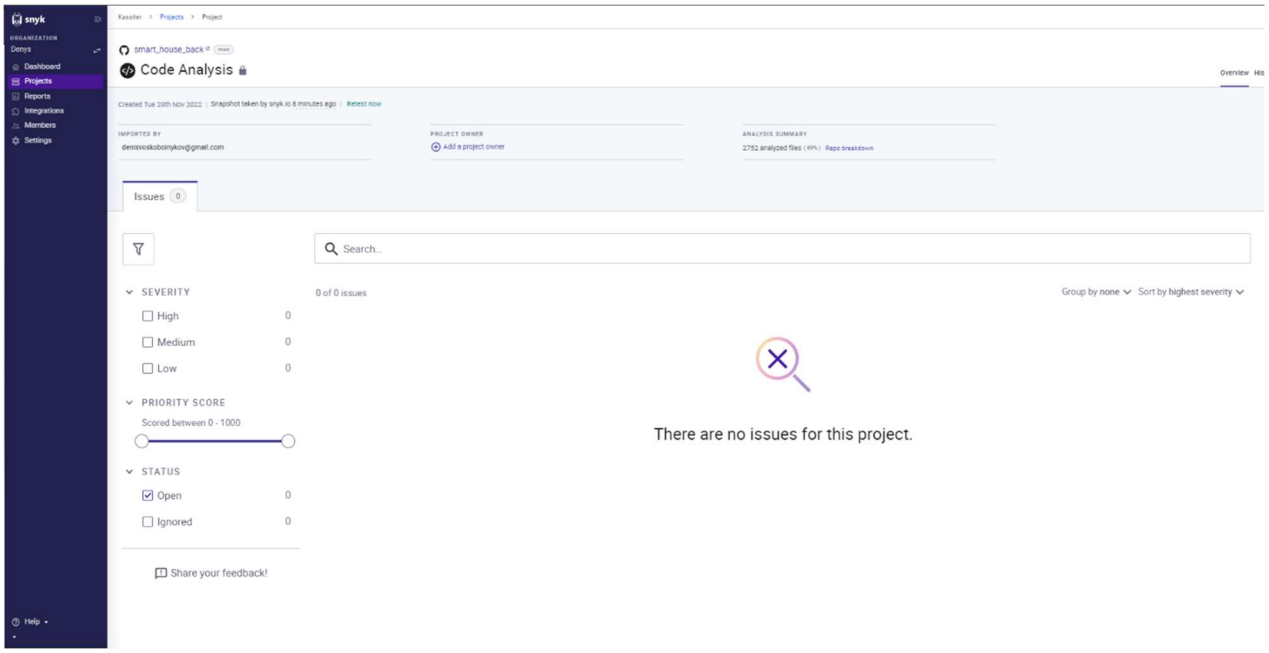

Рисунок 3.9 – Результат повторного сканування системи

ВИСНОВКИ

1) Завдяки застосуванню кращих засобів боротьби з вразливостями веб-застосунків з’являється реальна можливість позбутися критичних недоліків системи й забезпечити належний рівень безпеки кінцевого продукту, що, в свою чергу, зробить ПЗ більш привабливим для користувачів й дозволить ефективно розвиватися в умовах швидкорозвиваючогося ринку мережевих веб-програм. Регулярно слідкувати й своєчасно оновлювати безпеку системи, використовуючи при цьому новітні інструменти розробки та методи захисту є найважливішим кроком у створенні захищеного ПЗ.

2) Щоб розібратися з питаннями забезпечення захисту веб-застосунків від найпоширеніших критичних вразливостей системи та забезпечити кращу якість ПЗ було виконано дослідження предметної галузі. Розглянуто основні питання веб-застосунків, типову архітектуру систем, типи веб-застосунків, найпоширеніші вразливості безпеки й методи боротьби з ними, правила реалізації захищених ПЗ, сучасні методи контролю якості систем.

3) На основі проведеного аналізу предметної області та проведення дослідження вразливостей веб-застосунків й способів реалізації захищених систем проведено проектування застосунку, в ході якого було визначено наступні вимоги:

- загальні функціональні вимоги до системи;
- вимоги до користувацького інтерфейсу;
- вимоги до API системи;
- вимоги до механізмів захисту веб-застосунку.

Задля кращого розуміння проектованої системи й легшої подальшої її реалізації було створено UML діаграми, серед яких можна виділити діаграму послідовності реєстрації користувача й діаграму послідовності авторизації, котрі

мають за мету описати процеси реєстрації та авторизації. Реалізовано модель сутностей та зв’язків між ними(ER-модель).

4) Згідно з визначеними вимогами під час проектування системи було обрано технології розробки клієнту, серверу й БД веб-застосунку, котрі дозволяють реалізувати захищену систему з використанням кращих практик щодо забезпечення захисту веб-застосунку від найпоширеніших вразливостей та інших недоліків безпеки систем. Попри те, що обрані технології мають корисні вбудовані захисту, було реалізовані додаткові сучасні механізми захисту.

5) Реалізовану систему розумного будинку було протестовано різними підходами, серед яких мануальне тестування, end-to-end тестування й unit-тестування. Безпеку системи було перевірено сканером Snyk.

6) Отримані в ході виконання роботи результати доводять ефективність використаних новітніх механізмів захисту від найпоширеніших вразливостей, що було продемонстровано в рамках створення захищеної системи розумного будинку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Single-Page Apps vs Multiple-Page Web Apps. *Agilie*. URL: https://agilie.com/blog/single-page-apps-vs-multiple-page-web-apps (дата звернення: 3.10.2022)
2. Sbarski, Peter, and Sam Kroonenburg. Serverless architectures on AWS: with examples using Aws Lambda. Simon and Schuster, 2017.
3. What is PWA? *Vue Storefront*. URL: https://vuestorefront.io/pwa (дата звернення: 6.10.2022)
4. OWASP Top 10:2021. *OWASP*. URL: https://owasp.org/Top10 (дата звернення: 10.10.2022)
5. OWASP Top 10 Risks and How to Prevent Them. *Bright Security*. URL: https://brightsec.com/blog/owasp-top-10 (дата звернення: 11.10.2022)
6. Attributes of secure web application architecture. *Synopsys*. URL: https://www.synopsys.com/blogs/software-security/attributes-of-secure-web-application-architecture (дата звернення: 14.10.2022)
7. Static application security testing (SAST). *Invicti*. URL: https://www.invicti.com/learn/static-application-security-testing-sast (дата звернення: 14.10.2022)
8. Seth, Aishwarya. “Comparing Effectiveness and Efficiency of Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP) Tools.” (2022).
9. Definition of Runtime Application Self-Protection (RASP). *Gartner*. URL: https://www.gartner.com/en/information-technology/glossary/runtime-application-self-protection-rasp (дата звернення: 15.10.2022)
10. Горбенко І.Д., Горбенко Ю.І. Прикладна криптологія: Теорія. Практика. Застосування. Монографія. – Харків, ФОРТ, 2012, 880 с.

11. Audit Logging Overview. *Datadog*. URL: https://www.datadoghq.com/knowledge-center/audit-logging (дата звернення: 23.10.2022)
12. Are Your Passwords in the Green? *Hive Systems*. URL: https://www.hivesystems.io/blog/are-your-passwords-in-the-green (дата звернення: 23.10.2022)
13. Understanding Functional and Non-Functional Requirements in App Development. *Moveo Apps*. URL: https://www.moveoapps.com/blog/functional-and-non-functional-requirements-in-app-development (дата звернення: 26.10.2022)
14. User Interface Design Guidelines For Web Applications. *DesignyUp*. URL: https://www.designyup.com/7-user-interface-design-guidelines-for-web-applications (дата звернення: 27.10.2022)
15. What is an API: Definition, Types, Specifications, Documentation. *AltexSoft*. URL: https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation (дата звернення: 1.11.2022)
16. 7 Web Application Security Practices You Can Use. *Security Boulevard*. URL: https://securityboulevard.com/2021/10/7-web-application-security-practices-you-can-use (дата звернення: 5.11.2022)
17. Use Case Diagram Tutorial. *Creately*. URL: https://creately.com/blog/diagrams/use-case-diagram-tutorial (дата звернення: 7.11.2022)
18. Unified Modeling Language (UML) | Sequence Diagrams. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams (дата звернення: 8.11.2022)
19. Entity Relationship (ER) Diagram Model with DBMS Example. *Guru99*. URL: https://www.guru99.com/er-diagram-tutorial-dbms.html (дата звернення: 12.11.2022)

20. React.js for Web Applications: What a CTO Should Know before Development Begins. *RubyGarage*. URL: https://rubygarage.org/blog/react-js-for-web-applications (дата звернення: 12.11.2022)
21. Яремчук К., Воскобойников Д. Сучасні загрози та методи забезпечення безпеки веб-застосунків. *Комп'ютерні науки та кібербезпека*. 2022. №. 2. С. 26-32. doi: 10.26565/2519-2310-2022-2-03
22. How to use Node Js for Backend Web Development in 2023. *Simplilearn*. URL: https://www.simplilearn.com/tutorials/nodejs-tutorial/nodejs-backend (дата звернення: 15.11.2022)
23. Express web framework. *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs (дата звернення: 15.11.2022)
24. What Is MySQL And Why It Is Used? *Software Testing Help*. URL: https://www.softwaretestinghelp.com/what-is-mysql (дата звернення: 15.11.2022)
25. How To Use Sequelize with Node.js and MySQL. *DigitalOcean*. URL: https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql (дата звернення: 20.11.2022)
26. Manual testing. *Wikipedia The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Manual_testing (дата звернення: 23.11.2022)
27. What Is Automation Testing (Ultimate Guide To Start Test Automation). *Software Testing Help*. URL: https://www.softwaretestinghelp.com/automation-testing-tutorial-1 (дата звернення: 25.11.2022)
28. Unit testing. *Wikipedia The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Unit_testing (дата звернення: 25.11.2022)
29. What is End To End Testing? *BrowserStack*. URL: https://www.browserstack.com/guide/end-to-end-testing (дата звернення: 25.11.2022)

30. What is Snyk? *Snyk*. URL: https://snyk.io/what-is-snyk (дата звернення: 28.11.2022)

ДОДАТОК А

СПИСОК ПУБЛІКАЦІЙ МАГІСТРА

ISSN 2519-2310 CS&CS, Issue 2(22) 2022

УДК 681.04

СУЧАСНІ ЗАГРОЗИ ТА СПОСОБИ ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ ВЕБ-ЗАСТОСУНКІВ

Кирило Яремчук, Денис Воскобойников

Харківський національний університет імені В.Н. Каразіна, Харків, 61022, Україна
kir.yaremchuk@gmail.com, denisvoskobojnykov@gmail.com

Рецензент: Микола Карпінський, д.т.н., проф., університет Бельсько-Біла, Бельсько-Біла, Польща.
mkarpinski@ath.bielsko.pl

Надійшла: Жовтень 2022.

Анотація: *Складність розроблюваних веб-застосунків зростає з кожним роком, що, в свою чергу, робить важкоздійсненим забезпечення їхньої безпеки. Саме тому, доцільно приділяти особливу увагу критичним проблемам захисту програмного забезпечення. Вміння оцінювати ризики та запобігати вразливостям ще на етапі проектування продукту є вкрай важливою задачею, котра знижує потенційні складності при експлуатації застосунку. За останні роки кількість випадків витоку даних у всіх галузях ринку зменшилося, але, їх руйнівність стала значнішою. Серед усіх атак, атаки на веб-застосунки становлять більш ніж 50 відсотків. Згідно зі списком вразливостей OWASP Top Ten, в роботі розглянуто актуальні категорії вразливостей та напрямки атак на існуючі веб-застосунки. Було розглянуто ефективні способи їх запобігання. Наведені рекомендації щодо реалізації та підтримки захищеності додатків, розроблених з використанням бібліотеки ReactJS. Було виділено найпоширеніші загрози безпеки продуктів на базі React на протязі життєвого циклу додатку. Розглянуті основні способи оптимізації ReactJS.*

Ключові слова: *вразливість; веб-застосунки; загрози веб-застосунків; методи безпеки ReactJS*

1. Вступ

Завдяки тривалому часу свого розвитку, веб-застосунки почали являти собою дещо куди значніше, аніж просто сайти з контентом. На просторах мережі Інтернет с кожним днем з'являються все більш складні веб-застосунки за своїми цілями й можливостями, котрі пропонують нові рішення задля задоволення вимог споживачів в усіх галузях ринку. Веб-застосунки являють собою найбільш зручний та ефективний засіб для представлення інформації й надання послуг у мережі. Компанії з різнобічних галузей ринку продовжують створювати веб-застосунки для просування своїх товарів та послуг у Інтернеті, займаючи свої ніші у цифровому світі. Мобільні інструменти та веб-технології захопили вершину списку на довгі роки. Такі цифрові сервіси часто бувають критично значущими й потребують захисту задля забезпечення безпеки різного роду конфіденційної інформації. Галузь веб-технологій розвивається невпинними кроками, однак несе з собою і певні ризики безпеки: - забезпечення захисту значної кількості різнотипної інформації досягти досить складно. Недотримання вимог безпеки може загрожувати втратою ресурсів, а інколи й проблемами з законом [1]. На рис. 1 представлені тенденції злому систем безпеки у 2014-2021 роках [2].

Відповідно звіту *Risk Based Security* про тенденції порушення даних за 2021 рік, у 2020 та 2021 роках було втрачено 27,81 і 18,88 млрд записів, тоді як у 2019 році усього 4,681 млрд. записів [2]. Попри те, що кількість таких випадків зменшилася у 2020 та 2021 роках, втрати даних стали більш масштабними. В цілому, можна спостерігати відсутність кореляції між кількістю випадків втрати даних та кількістю втрачених даних, що говорить про те, що навіть одне малозначне порушення безпеки може спричинити серйозні наслідки. Щоб на практиці знизити можливі ризики безпеки, потрібно ретельно контролювати усі потенційні загрози та уважно дотримуватися існуючих стандартів інформаційної безпеки (ІБ). На рис. 2 представлено огляд основних тенденцій атак (зломів) по галузям економіки у 2021 році [2].

ДОДАТОК Б

ЛІСТИНГ ВЕБ-ЗАСТОСУНКУ

```
const express = require('express');

const { isHomeRecordExist } = require('../middlewares');


const homesRouter = require('./homesRouterv2');
const devicesRouter = require('./devicesRouterv2');

const {
  loginUser,
  registerUser,
  logout,
} = require('./controller');

const v2 = express.Router();

v2.route.post('/login', loginUser);
v2.route.post('/register', registerUser);
v2.route.get('/logout', logout);

v2.use('/homes', homesRouter);
v2.use('/homes/:homeid/devices', isHomeRecordExist, devicesRouter);

module.exports = v2;
```

Лістинг Б.1 – Маршрутизатор

```
const app = require('express');

const device = require('../controllers/v2/device');
const validationSchema = require('../helpers/schemas');
const { validationMiddleware, getDefaultPagParams } =
require('../middlewares');

const devicesRouter = app.Router();

devicesRouter
  .route('/')
  .post(
    validationMiddleware(validationSchema.device, 'body'),
    device.postDevice
  )
  .get(
    validationMiddleware(validationSchema.query, 'query'),
    getDefaultPagParams,
```

```
    device.getDevices
  );

devicesRouter
  .route('/:id')
  .get(device.getDeviceByPk)
  .put(
    validationMiddleware(validationSchema.device, 'body'),
    device.updateDevice
  )
  .delete(device.deleteDevice);

module.exports = devicesRouter;
```

Лістинг Б.2 – Маршрутизатор пристроїв

```
const app = require('express');

const home = require('../controllers/v2/home');
const validationSchema = require('../helpers/schemas');
const { validationMiddleware } = require('../middlewares');

const homesRouter = app.Router();

homesRouter
  .route('/')
  .post(validationMiddleware(validationSchema.homePOST, 'body'),
home.postHome)
  .get(home.getAllHomes);

homesRouter
  .route('/:homeid')
  .get(home.getHome)
  .delete(home.deleteHome)
  .put(validationMiddleware(validationSchema.homePUT, 'body'),
home.updateHome);

module.exports = homesRouter;
```

Лістинг Б.3 – Маршрутизатор будинків

```
module.exports = (sequelize, Sequelize) => {
  const Device = sequelize.define(
    'device',
    {
      name: {
        type: Sequelize.STRING,
      },
      status: {
        type: Sequelize.BOOLEAN,
      },
      type: {
        type: Sequelize.STRING,
```

```
      },
      image: {
        type: Sequelize.STRING,
      },
    },
    { timestamps: false }
  );
  return Device;
};

module.exports = (sequelize, Sequelize) => {
  const House = sequelize.define(
    'house',
    {
      name: {
        type: Sequelize.STRING,
      },
    },
    { timestamps: false }
  );
  return House;
};

module.exports = (sequelize, Sequelize) => {
  const Mode = sequelize.define(
    'mode',
    {
      type: {
        type: Sequelize.STRING,
      },
    },
    { timestamps: false }
  );
  return Mode;
};

module.exports = (sequelize, Sequelize) => {
  const ModeList = sequelize.define(
    'mode_list',
    {
      name: {
        type: Sequelize.STRING,
        unique: true,
      },
    },
    { timestamps: false }
  );
  return ModeList;
};

module.exports = (sequelize, Sequelize) => {
  const ModeList = sequelize.define(
```

```
    'mode_modeList',
    {
      modeId: {
        type: Sequelize.INTEGER,
      },
      modeListId: {
        type: Sequelize.INTEGER,
      },
      status: {
        type: Sequelize.BOOLEAN,
      },
    },
    { timestamps: false }
  );
  return ModeList;
};

module.exports = (sequelize, Sequelize) => {
  const Range = sequelize.define(
    'range',
    {
      min: {
        type: Sequelize.INTEGER,
      },
      max: {
        type: Sequelize.INTEGER,
      },
      current: {
        type: Sequelize.INTEGER,
      },
      step: {
        type: Sequelize.INTEGER,
      },
      type: {
        type: Sequelize.STRING,
      },
    },
    { timestamps: false }
  );
  return Range;
};
```

Лістинг Б.4 – Моделі бази даних

```
const router = require('express').Router()
const {body, validationResult} = require('express-validator')
module.exports = router
const User = require('../db/models/users')

router.post(
  '/signup',
  body('email').isEmail(),
  body('password').isLength({min: 6})
```

```
  .matches('[0-9]').withMessage('Password Must Contain a Number')
  .matches('[A-Z]').withMessage('Password Must Contain an
Uppercase Letter')
  .trim().escape()

  async (req, res, next) => {
    const errors = validationResult(req)

    try {
      if (!errors.isEmpty() && errors.errors[0].param === 'email')
{
        return res.status(400).send('Invalid email address. Please
try again.')
      }
      if (!errors.isEmpty() && errors.errors[0].param ===
'password') {
        return res
          .status(400)
          .send('Password must be longer than 6 characters.')
      }
      const user = await User.create(req.body)
      req.login(user, err => (err ? next(err) : res.json(user)))
    } catch (err) {
      next(err)
    }
  }
)
```

Лістинг Б.5 – Валідація пошти й паролю

```
const express = require('express')
const cors = require('cors')
const app = express()

const whitelist = process.env.CORS_WHITELIST

const corsOptions = {
  origin: function (origin, callback) {
    if (whitelist.indexOf(origin) !== -1) {
      callback(null, true)
    } else {
      callback(new Error('Not allowed by CORS'))
    }
  }
}

app.get('/devices/:id', cors(corsOptions), device.getDevice))
```

Лістинг Б.6 – Використання файлу оточення

```
bcrypt
  .genSalt(10)
  .then(salt => {
```

```
    return bcrypt.hash(password, salt)
  })
```

Лістинг Б.7 – Хешування паролю

```
const jwt  =  require('jsonwebtoken');

const secret = process.env.secret;

JWTGenerator = (id, res, req) => {
  const expiresIn = 24 * 60 * 60;
    const accessToken = jwt.sign({
      id: id
    }, SECRET_KEY, {
      expiresIn: expiresIn
    });
    req.session.token = accessToken;
    req.session.save(function(err) {
      if(!err){
        res.status(200).send({
          "id_user": id,
          "token": accessToken,
          "expires_in": expiresIn
        });
      }
    });
}

exports.JWTGenerator = JWTGenerator;
```

Лістинг Б.8 – Генерація JWT токена

```
module.exports = DevicePolicy

function DevicePolicy (user, device) {
  this.user = user
  this.device = device
}

DevicePolicy.prototype.edit = function (done) {
  if (!this.user || !this.device) return done(undefined, false)
  done(undefined, this.user.id === this.device.house.user_id)
}
```

Лістинг Б.9 – Механізм авторизації

```
const bouncer = require('express-bouncer');
bouncer.whitelist.push('127.0.0.1');

bouncer.blocked = function (req, res, next, remaining) {
    res.send(429, "Too many requests have been made. Please wait "
+ remaining/1000 + " seconds.");
};
```

```
app.post("/signin", bouncer.block, function(req, res) {
    if (LoginFailed){  }
    else {
        bouncer.reset( req );
    }
});
```

Лістинг Б.10 – Захист від атак «грубої сили»

```
const passwordValidator = require('password-validator');

const schema = new passwordValidator();

schema
  .is().min(8)
  .is().max(100)
  .has().uppercase()
  .has().lowercase();
```

Лістинг Б.11 – Валідація паролю на клієнті

```
const connection = require('./db');

app.post("/topics", (request, response) => {
    const query = `SELECT * FROM Devices WHERE id =
${connection.escape(request.body.id)}`;

    connection.query(query).then((topics) => {
       ...
       response.send(topics);
    });
});
```

Лістинг Б.12 – Використання безпечних параметрів для SQL запитів

```
const express = require('express')
const fs = require('fs')
const morgan = require('morgan')
const path = require('path')

const app = express()

const accessLogStream = fs.createWriteStream(path.join(__dirname,
'access.log'), { flags: 'a' })

app.use(morgan('combined', { stream: accessLogStream }))
```

Лістинг Б.13 – Логування етапів виконання системи

```
require('dotenv').config();

db.connect({
  host: process.env.DB_HOST,
  username: process.env.DB_USER,
  password: process.env.DB_PASS
```

```
})
```

Лістинг Б.14 – Використання файлу оточення

```
const services = require('../src/services/v1');
const { filterByType, searchBySubname } =
require('../src/services/v1/device');

describe('services', () => {
  it('updateItemById() updates array by id', () => {
    const arr = [
      { id: 1, name: 'abc' },
      { id: 2, name: 'dfe' },
    ];

    const expRes = {
      updatedItems: [
        { id: 1, name: 'abc' },
        { id: 2, name: 'test' },
      ],
      wasUpdated: true,
    };

    const res = services.updateItemById(arr, 2, { name: 'test' });

    expect(expRes).toEqual(res);
  });

  it('updateItemById() does not update array by incorrect id', ()
=> {
    const arr = [
      { id: 1, name: 'abc' },
      { id: 2, name: 'dfe' },
    ];

    const expRes = {
      updatedItems: [
        { id: 1, name: 'abc' },
        { id: 2, name: 'dfe' },
      ],
      wasUpdated: false,
    };

    const res = services.updateItemById(arr, 3, { name: 'test' });

    expect(expRes).toEqual(res);
  });

  it('getItemById() returns required item from array by id', () =>
{
    const arr = [
      { id: 1, name: 'abc' },
      { id: 2, name: 'dfe' },
```

```
    ];

    const expRes = { id: 1, name: 'abc' };

    const res = services.getItemById(1, arr);

    expect(expRes).toEqual(res);
  });

  it('getItemById() returns undefined from array by incorrect id',
() => {
    const arr = [
      { id: 1, name: 'abc' },
      { id: 2, name: 'dfe' },
    ];

    const expRes = undefined;

    const res = services.getItemById(3, arr);

    expect(expRes).toEqual(res);
  });

  it('deleteItemById() deletes item from array', () => {
    const arr = [
      { id: 1, name: 'abc' },
      { id: 2, name: 'dfe' },
    ];

    const expRes = { updatedItems: [{ id: 2, name: 'dfe' }],
wasUpdated: true };

    const res = services.deleteItemById(arr, 1);

    expect(expRes).toEqual(res);
  });

  it('deleteItemById() does not delete item from array because of
incorrect id', () => {
    const arr = [
      { id: 1, name: 'abc' },
      { id: 2, name: 'dfe' },
    ];

    const expRes = {
      updatedItems: [
        { id: 1, name: 'abc' },
        { id: 2, name: 'dfe' },
      ],
      wasUpdated: false,
    };
```

```
    const res = services.deleteItemById(arr, 15);

    expect(expRes).toEqual(res);
  });

  it('filterByType() filters array by type correctly', () => {
    const arr = [
      { id: 1, type: 'oven' },
      { id: 2, type: 'robot' },
      { id: 3, type: 'oven' },
      { id: 4, type: 'robot' },
      { id: 5, type: 'tv' },
      { id: 6, type: 'OVEN' },
      { id: 7, type: 'RoBoT' },
    ];

    const expRes = [
      { id: 2, type: 'robot' },
      { id: 4, type: 'robot' },
      { id: 7, type: 'RoBoT' },
    ];

    const res = filterByType(arr, 'RObOT');

    expect(expRes).toEqual(res);
  });

  it('searchBySubname() filters array by subname correctly', () =>
{
    const arr = [
      { id: 1, name: '12oven' },
      { id: 2, name: 'robot' },
      { id: 3, name: 'oven-Cool' },
      { id: 4, name: 'robot-bad' },
      { id: 5, name: 'tv' },
      { id: 6, name: 'cool-OVEN132' },
      { id: 7, name: 'RoBoT' },
    ];

    const expRes = [
      { id: 1, name: '12oven' },
      { id: 3, name: 'oven-Cool' },
      { id: 6, name: 'cool-OVEN132' },
    ];

    const res = searchBySubname(arr, 'OveN');

    expect(expRes).toEqual(res);
  });

  it('searchBySubname() filters array by subname correctly', () =>
{
```

```
    const arr = [
      { id: 1, name: '12oven' },
      { id: 2, name: 'robot' },
      { id: 3, name: 'oven-Cool' },
      { id: 4, name: 'robot-bad' },
      { id: 5, name: 'tv' },
      { id: 6, name: 'cool-OVEN132' },
      { id: 7, name: 'RoBoT' },
    ];

    const expRes = [
      { id: 1, name: '12oven' },
      { id: 3, name: 'oven-Cool' },
      { id: 6, name: 'cool-OVEN132' },
    ];

    const res = searchBySubname(arr, 'OveN');

    expect(expRes).toEqual(res);
  });

  it('paginate() returns correct information', () => {
    const arr = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l'];

    const expRes1 = {
      data: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l'],
      totalItems: 12,
      totalPages: 1,
      page: 1,
      perPage: 30,
    };

    const expRes2 = {
      data: ['a', 'b', 'c', 'd'],
      totalItems: 12,
      totalPages: 3,
      page: 1,
      perPage: 4,
    };

    const expRes3 = {
      data: ['k', 'l'],
      totalItems: 12,
      totalPages: 3,
      page: 3,
      perPage: 5,
    };

    const res1 = services.paginate(arr, 2, 30);
    const res2 = services.paginate(arr, 1, 4);
```

```
    const res3 = services.paginate(arr, 4, 5);

    expect(expRes1).toEqual(res1);
    expect(expRes2).toEqual(res2);
    expect(expRes3).toEqual(res3);
  });
});
```

Лістинг Б.15 – Юніт тести

```
const request = require('supertest');

const app = require('../src/app');

describe('Home endpoints', () => {
  it('should respond with 422 status code because validation
error', async () => {
    const res = await request(app).post('/api/v1/homes').send({
      name: 'third',
      devices: [],
      sad: 'sd',
    });

    expect(res.statusCode).toEqual(422);
  });

  it('should create a new home', async () => {
    const res = await request(app).post('/api/v1/homes').send({
      name: 'third',
      devices: [],
    });

    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveProperty('id');
  });

  it('should get 1 home', async () => {
    const res = await request(app).get('/api/v1/homes/2');

    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveProperty('id');
  });

  it('should get all homes', async () => {
    const res = await request(app).get('/api/v1/homes');

    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveLength(2);
  });

  it('should update a home', async () => {
    const res = await request(app).put('/api/v1/homes/2').send({
      name: 'good',
```

```
    });

    expect(res.statusCode).toEqual(200);
  });

  it('should delete a home', async () => {
    const res = await request(app).delete('/api/v1/homes/2');

    expect(res.statusCode).toEqual(200);
  });

  it('should respond with status code 404 if home is not found',
async () => {
    const res = await request(app).get('/api/v1/homes/2');

    expect(res.statusCode).toEqual(404);
  });
});

describe('Device endpoints', () => {
  it('should respond with 422 status code because validation
error', async () => {
    const res = await request(app)
      .post('/api/v1/homes/1/devices')
      .send({
        name: 'Oven',
        status: true,
        type: 'oven',
        image: '',
        temp: {
          max: 0,
          current: 0,
          step: 0,
        },
        modes: [],
        currentMode: '',
      });

    expect(res.statusCode).toEqual(422);
  });

  it('should create a new device', async () => {
    const res = await request(app)
      .post('/api/v1/homes/1/devices')
      .send({
        name: 'Oven',
        status: true,
        type: 'oven',
        image: '',
        temp: {
          min: 0,
          max: 0,
```

```
          current: 0,
          step: 0,
        },
        modes: [],
        currentMode: '',
      });

    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveProperty('id');
  });

  it('should get 1 device', async () => {
    const res = await
request(app).get('/api/v1/homes/1/devices/1');

    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveProperty('id');
  });

  it('should get all devices', async () => {
    const res = await request(app).get('/api/v1/homes/1/devices');

    expect(res.statusCode).toEqual(200);
    expect(res.body).toHaveProperty('data');
    expect(res.body).toHaveProperty('page');
  });
  it('should update a device', async () => {
    const res = await request(app)
      .put('/api/v1/homes/1/devices/1')
      .send({
        name: 'Oven',
        status: false,
        type: 'oven',
        image: '',
        temp: {
          min: 0,
          max: 0,
          current: 0,
          step: 0,
        },
        modes: [],
        currentMode: '',
      });

    expect(res.statusCode).toEqual(200);
  });

  it('should delete a device', async () => {
    const res = await
request(app).delete('/api/v1/homes/1/devices/1');

    expect(res.statusCode).toEqual(200);
```

```
  });

  it('should respond with status code 404 if device is not found',
async () => {
    const res = await
request(app).get('/api/v1/homes/1/devices/1');

    expect(res.statusCode).toEqual(404);
  });
});
```

Лістинг Б.16 – End-to-end тести