МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ЗАСОБАМИ ANGULAR, AWS LAMBDA»**

Виконав: студент 4 курсу, групи 6.1211-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
О.С. Удод
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Кривохата А.Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри комп'ютерних наук, професор, д.т.н. Шило Г.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Удоду Олегу Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи управління завданнями засобами Angular, AWS Lambda

керівник роботи Кривохата Анастасія Григорівна, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз сучасних рішень та обраних технологій розробки.
2. Аналіз вимог та проєктування.
3. Програмна реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

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

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.01.2025	
2.	Збір вихідних даних.	17.02.2025	
3.	Обробка методичних та теоретичних	03.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	24.04.2025	
5.	Розробка третього розділу.	15.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	19.06.2025	

Студент _____ О.С. Удод
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.Г. Кривохата
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка системи управління завданнями засобами Angular, AWS Lambda»: 68 с., 25 рис., 17 джерел.

АВТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ, БЕЗСЕРВЕРНА АРХІТЕКТУРА, БЕКЕНД, ВЕБЗАСТОСУНОК, СИСТЕМА УПРАВЛІННЯ ЗАВДАННЯМИ, ФРОНТЕНД, ХМАРНІ СЕРВІСИ, ANGULAR, AWS LAMBDA, JWT, KANBAN-ДОШКА, POSTGRESQL.

Об’єкт дослідження – процес розробки та впровадження системи управління завданнями з використанням вебтехнологій та хмарних сервісів.

Предмет дослідження – фреймворк Angular, хмарні технології, AWS Lambda.

Мета роботи: розробка ефективної, масштабованої та безпечної системи управління завданнями з використанням фреймворку Angular для клієнтської частини та сервісу AWS Lambda для реалізації серверної логіки.

Метод дослідження – аналіз аналогічних рішень, проєктування з використанням об’єктно-орієнтованого підходу, реалізація програмного забезпечення та його тестування.

У роботі проаналізовано системи управління завданнями, обґрунтовано вибір фреймворку Angular для клієнтської частини та AWS Lambda для серверної логіки. Розроблено архітектуру системи з компонентною структурою фронтенду та безсерверною інфраструктурою бекенду. Реалізовано основні компоненти, як-от: інтерфейс користувача, механізми автентифікації на основі JWT та логіку обробки запитів. Здійснено тестування основних компонентів сучасними засобами, зокрема Jest, Postman та Artillery. Результатом роботи є протестована система управління завданнями, що відповідає вимогам до вебзастосунків та може впроваджуватись для покращення процесів управління проєктами.

SUMMARY

Bachelor’s qualifying paper “Development of a Task Management System Using Angular, AWS Lambda”: 68 pages, 25 figures, 17 references.

AUTHENTICATION, AUTHORIZATION, SERVERLESS ARCHITECTURE, BACKEND, WEB APPLICATION, TASK MANAGEMENT SYSTEM, FRONTEND, CLOUD SERVICES, ANGULAR, AWS LAMBDA, JWT, KANBAN BOARD, POSTGRESQL.

Object of the study – the process of development and implementation of a task management system using web technologies and cloud services.

Subject of the study – Angular framework, cloud technologies, AWS Lambda.

Aim of the study: development of an effective, scalable, and secure task management system using the Angular framework for the client side and AWS Lambda service for implementing server logic.

The methods of research – analysis of similar solutions, design using object-oriented approach, software implementation and testing.

The work analyzes task management systems, justifies the choice of Angular framework for the client side and AWS Lambda for server logic. The system architecture with component structure of the frontend and serverless infrastructure of the backend was developed. The main components were implemented, such as: user interface, JWT-based authentication mechanisms, and request processing logic. Testing of the main components was performed with modern tools, including Jest, Postman, and Artillery. The result of the work is a tested task management system that meets the requirements for web applications and can be implemented to improve project management processes.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 7
1 Аналіз сучасних рішень та технологій..... 9
1.1 Огляд сучасних систем управління завданнями..... 9
1.2 Аналіз та обґрунтування вибору технологій Angular для фронтенд-розробки 12
1.3 Можливості AWS Lambda та інших сервісів AWS для реалізації серверної частини..... 15
1.4 Висновки до розділу 1 18
2 Проєктування системи управління завданнями..... 19
2.1 Аналіз функціональних та нефункціональних вимог до системи 19
2.2 Концептуальне проєктування інтерфейсу та варіантів використання.21
2.3 Логічне проєктування архітектури системи та структури даних 27
2.4 Фізичне проєктування компонентів та схеми розгортання 36
2.5 Висновки до розділу 2 39
3 Реалізація та тестування системи 40
3.1 Реалізація компонентів клієнтської частини на Angular..... 40
3.2 Реалізація серверної логіки з використанням AWS Lambda 47
3.3 Система автентифікації та авторизації 53
3.4 Тестування компонентів та інтеграційне тестування системи..... 57
3.5 Висновки до розділу 3 65
Висновки 66
Перелік посилань..... 67

ВСТУП

Сучасний розвиток засобів створення вебзастосунків призводить до того, що з’являються нові можливості для ефективної розробки та реалізації систем управління завданнями у вебсередовищі.

Прикладом таких систем є популярні платформи управління проєктами та завданнями, такі як Wrike, JIRA, Trello, де користувачі можуть створювати проєкти, додавати завдання, встановлювати пріоритети, відстежувати прогрес виконання та обмінюватись коментарями.

Отже, актуальною задачею є розробка системи управління завданнями “TaskManager”.

Метою кваліфікаційної роботи є розробка системи управління завданнями засобами Angular та AWS Lambda.

Задачі, які необхідно розв’язати для досягнення поставленої мети:

– проаналізувати існуючі рішення та технології розробки;
– сформулювати вимоги до системи управління завданнями;
– спроєктувати архітектуру системи та структуру даних;
– реалізувати компоненти клієнтської частини на Angular;
– розробити серверну логіку з використанням AWS Lambda;
– створити систему автентифікації та авторизації;
– протестувати компоненти та провести інтеграційне та навантажувальне тестування.

Структурно робота складається з трьох розділів. У першому розділі проаналізовано існуючі системи управління завданнями (Wrike, JIRA, OpenProj), обґрунтовано вибір фреймворку Angular для фронтенд-розробки та досліджено можливості AWS Lambda та інших сервісів AWS для реалізації серверної частини. У другому розділі виконано аналіз функціональних та нефункціональних вимог до системи, розроблено концептуальну модель з діаграмами прецедентів та макетами інтерфейсу, створено логічну архітектуру системи та структуру бази даних, а також представлено фізичне проєктування

компонентів та схеми розгортання. У третьому розділі реалізовано компоненти клієнтської частини на Angular (включаючи форми авторизації, реєстрації та дошку завдань), розроблено серверну логіку з використанням AWS Lambda, створено систему автентифікації та авторизації на основі JWT, а також проведено комплексне тестування компонентів та інтеграційне тестування системи.

1 АНАЛІЗ СУЧАСНИХ РІШЕНЬ ТА ТЕХНОЛОГІЙ

1.1 Огляд сучасних систем управління завданнями

У сфері розробки програмного забезпечення та управління проєктами спостерігається значне розгалуження систем управління завданнями. Це призвело до появи значної кількості різноманітних систем, кожне з яких пропонує свій унікальний набір функцій, підтримуваних методології (Agile, Kanban та Waterfal) та можливостей та власний підхід до організації робочого процесу (від простих однорівневих до комплексних та декомпозованих у багато рівнів задач). Розгалуження систем створює необхідність детального аналізу існуючих варіантів, щоб зрозуміти їхні функціональні та нефункціональні можливості, особливості інтеграції, масштабованість та придатність до різних типів команд і проєктів, які суттєво відрізняються за своєю спрямованістю, складністю управління та специфікою бізнес-процесів.

Визначимо та наведемо аналіз особливостей аналогів розроблюваного програмного забезпечення. Найпопулярнішими на поточний час системами є:

– Wrike;
– Worksection;
– Asana;
– Trello;
– Microsoft Planner;
– JIRA;
– Microsoft Project;
– OpenProj [1, 2].

Проаналізуємо аналогічні (повністю або частково) рішень із наведеного вище переліку систем.

Wrike – це платформа для спільного управління роботою, яка розроблена для оптимізації планування проєктів, відстеження та виконання завдань, та надає інструменти для організації робочих процесів, управління ресурсами та

покращення командної співпраці, що робить її придатною для різноманітних бізнес-процесів. Налаштовуванні функції та інтеграції дозволяють командам адаптувати платформу під свої задачі.

Переваги Wrike:

– налаштування функцій для потреб різних команд, включаючи специфічні рішення для маркетингових груп та команд професійних послуг;
– можливість встановлення залежностей між завданнями з автоматичним коригуванням термінів, що мінімізує затримки;
– вбудовані інструменти для узгодження та затвердження активів клієнтами безпосередньо у платформі;
– відсутність прив’язки системи до однієї сфери користувачів.

Недоліки Wrike:

– висока вартість для малих команд (орієнтовно від $10 за користувача на місяць з мінімумом 2 місць);
– комплексна крива навчання через широкий спектр функцій;
– потенційно надмірна функціональність для команд з простими потребами управління завданнями;
– обмеження у вільній версії, які можуть бути недостатніми для повноцінного використання [2, 3].

Jira (JIRA) – це платформа для управління проєктами та відстеження завдань, розроблена компанією Atlassian. Система призначена для технічних команд, які працюють за гнучкими методологіями, дозволяючи планувати спринти, розподіляти завдання та відстежувати помилки в процесі розробки програмного забезпечення.

Переваги JIRA:

– система відстеження помилок та проблем, що дозволяє документувати, призначати та вирішувати технічні питання;
– інтеграція з іншими інструментами розробки програмного забезпечення, зокрема Bitbucket, GitHub та Jenkins;

– гнучкість робочих процесів з можливістю налаштування станів завдань, полів та правил переходу відповідно до специфіки команди;
– розширені можливості для роботи за Agile-методологіями, включаючи Scrum та Kanban дошки, з інструментами для планування спринтів.

Недоліки JIRA:

– складність налаштування та адміністрування, що потребує значних ресурсів для початкового впровадження;
– відносно висока вартість при масштабуванні, особливо для великих організацій з багатьма користувачами;
– надмірна функціональність для невеликих проєктів та нетехнічних команд, що може знижувати продуктивність [4].

OpenProj – це безкоштовна програма для управління проєктами з відкритим кодом, яка пропонує альтернативу комерційним рішенням, таким як Microsoft Project. Вона дозволяє користувачам створювати проєктні плани, відстежувати ресурси, візуалізувати часові рамки проєктів та керувати залежностями між завданнями.

Переваги OpenProj:

– безкоштовна альтернатива з відкритим кодом до дорогих комерційних інструментів управління проєктами, що зменшує витрати на програмне забезпечення;
– сумісність із файлами Microsoft Project, що дозволяє безперешкодно обмінюватися проєктними планами з користувачами різних систем;
– основний функціонал управління проєктами, включаючи діаграми Ганта (Гантта), графіки мережевих задач та структури розбивки робіт (WBS);
– можливість отримання детальної статистики за різних фільтрів.

Недоліки OpenProj:

– обмежені функції оновлення та технічної підтримки порівняно з комерційними альтернативами;
– менш інтуїтивний інтерфейс користувача;

- відсутність деяких передових функцій, які присутні в сучасних інструментах управління проєктами, таких як розширені можливості співпраці чи хмарна синхронізація;
- проблеми зі стабільністю при роботі з особливо великими чи складними проєктними планами;
- проблема із кросплатформенністю використання системи в операційних системах, інакших від Windows [5].

Отже, сучасний ринок систем управління завданнями характеризується розгалуженням рішень із різноманітними функціональними можливостями та методологічними підходами, що дозволяє організаціям обирати інструменти відповідно до специфіки їхніх проєктів, проте вимагає ретельного аналізу для визначення оптимального варіанту з урахуванням визначених компаніями критеріїв.

1.2 Аналіз та обґрунтування вибору технологій Angular для фронтенд-розробки

При розробці фронтенд частини програмного забезпечення вибір фреймворку відіграє критичну роль у забезпеченні ефективності, масштабованості та зручності використання кінцевого продукту. В умовах сучасного розвитку вебтехнологій існує багато фреймворків і бібліотек, кожен з яких має свої особливості, переваги та недоліки. Після ретельного аналізу наявних рішень, було прийнято рішення використовувати Angular як основну технологію розробки клієнтської частини системи управління задачами. У даному розділі буде проаналізовано характеристики Angular, обґрунтовано вибір даної технології з урахуванням специфіки проєкту та проведено порівняльний аналіз з популярними фреймворками.

По-перше, наведемо переваги Angular:

- компонентно-орієнтовна архітектура (структура фреймворку базується на компонентному підході, що сприяє створенню

модульних та перевикористовуваних частин інтерфейсу, які нескладно тестувати);

- система впровадження залежностей (вбудована система DI спрощує управління залежностями між компонентами та сервісами тим самим полегшує тестування та спряє підвищенню модульності коду);
- двостороннє зв’язування даних (фреймворк пропонує наведений механізм, який автоматично синхронізує модель та представлення, зменшуючи кількість шаблонного коду);
- комплексна платформа розробки (Angular надає повноцінну платформу з широким набором вбудованих інструментів, включаючи системи маршрутизації, HTTP-клієнт, форми, анімації, що дозволяє розробникам не витрачати час на інтеграцію сторонніх рішень);
- мова програмування TypeScript як основна мова розробки (забезпечує статичну типізацію, виявлення помилок на етапі компіляції та покращену підтримку IDE);
- довгострокова підтримка (фреймворк має стабільний цикл випуску нових версій і підтримки, забезпечуючи передбачуваність оновлень та довгострокову підтримку LTS від компанії Google) [6, 7].

Незважаючи на переваги наведеного фреймворку, Angular містить обмеження та недоліки, які потрібно враховувати для розробки якісного програмного продукту:

- розмір бандлу (стандартний розмір Angular-додатків більший порівняно з додатками на React або Vue.js, що може вплинути на швидкість першого завантаження проєкту);
- обмеженість у гнучкості (чітка вираженість структури проєкту може обмежувати свободу вибору рішень розробниками у певних рішеннях);
- складність оптимізації продуктивності (необхідність великих зусиль та глибокого розуміння внутрішніх механізмів фреймворку для покращення продуктивності проєкту) [8, 9].

Проаналізувавши загальні характеристики Angular та врахувавши особливості розроблюваної системи управління задачами, можна виділити ряд ключових переваг даного фреймворку, які роблять його оптимальним вибором саме для розробки системи управління завданнями. Специфіка систем управління завданнями, враховуючи її складність структури даних, необхідність інтерактивного інтерфейсу та високим вимогам до безпеки, знаходить відображення в архітектурних та інструментальних особливостях Angular:

– модульність та структурованість (вимога чіткої організації програмного коду через велику кількість взаємопов’язаних компонентів);
– робота з формами (значна частина функціоналу системи критично пов’язана з обробкою форм, наприклад, створення задач, редагування параметрів та фільтрація; Angular надає інструментарій для вирішення проблем з формами);
– керування станом (система управління задачами вимагає ефективного управління складними станами додатку, тому RxJS, який інтегровано в Angular, забезпечив реактивний підхід до обробки даних та подій, що особливо корисно при реалізації функціоналу реального часу та синхронізації змін між користувачами);
– безпека та автентифікація (фреймворк надає вбудовані механізми для захисту від поширених властивостей та інструменти для реалізації автентифікації та авторизації) [10].

При виборі технології для розробки системи управління завданнями важливо провести порівняльний аналіз з іншими популярними фреймворками на поточний час, які потенційно підходили б для розробки системи управління завданнями. На 2025 рік виділяють п’ять наступних популярних фреймворків: React, VueJS, Angular, Svelte та Solid.js [11].

Наведемо результати порівняння з двома найпопулярнішими фреймворками з наведеного списку. React пропонує гнучкий підхід з мінімальною структурою та фокусом на компонентах користувацького

інтерфейсу, але поступається Angular у комплексності та готовності рішення «з коробки», що для системи управління задачами означає необхідність інтегрувати додаткові бібліотеки для маршрутизації, управління станом та форм. Vue не забезпечує глибини TypeScript-інтеграції та корпоративного рівня архітектури, які надає Angular, що особливо важливо при розробці масштабованої системи управління задачами з багатьма користувачами та функціями [12].

Таким чином, вибір Angular для розробки системи управління задачами є обґрунтованим рішенням, що базується на його комплексній архітектурі, вбудованих інструментах та TypeScript-інтеграції, які забезпечують необхідну структурованість, безпеку та масштабованість для реалізації функціональних вимог системи.

1.3 Можливості AWS Lambda та інших сервісів AWS для реалізації серверної частини

Для реалізації серверної частини системи управління задачами було проведено детальний аналіз хмарних сервісів Amazon Web Services (AWS), які надають можливості для створення масштабованих, безпечних, та економічно ефективних рішень. При розробці програмного забезпечення, найактивніше було залучено наступні сервіси:

- AWS Lambda;
- Amazon VPC;
- Amazon EC2;
- RDS PostgreSQL;
- AWS Secrets Manager;
- Amazon API Gateway.

Реалізація серверної частини потребує детального аналізу широко залучених сервісів, тому по кожному із наведених сервісів вище наведено короткі результати дослідження можливостей.

AWS Lambda – це сервіс обчислень без серверів, який дозволяє запускати код без необхідності налаштування та управління серверами. Для системи управління задачами Lambda виконує роль основного обчислювального механізму, обробляючи запити користувачів, взаємодіючи з базою даних та іншими сервісами AWS. Ключовими перевагами використання Lambda для проєкту є автоматичне масштабування відповідно до навантаження, модель оплати лише за фактично виконані обчислення та підтримка різних мов програмування, включаючи Node.js. Завдяки Lambda надано можливість розробити мікросервісну архітектуру, де кожна функція відповідає за конкретну бізнес-логіку системи управління задачами, такі як створення завдань, призначення відповідальних, генерація звітів та сповіщення користувачів [13].

Amazon VPC – це сервіс, який дозволяє створити ізольовану приватну мережу в хмарі AWS з визначеним діапазоном IP-адрес, налаштуваннями підмереж та таблицями маршрутизації. Для системи управління задачами VPC забезпечує надійний рівень мережевої ізоляції та безпеки, контролюючи доступ до ресурсів та захищаючи дані від несанкціонованого доступу [14]. Використання VPC дозволило реалізувати багаторівневу архітектуру, де публічні сервіси, такі як API Gateway, розміщені в публічних підмережах, а бази даних та інші критичні компоненти – в приватних підмережах з контрольованим доступом.

Amazon EC2 – це сервіс віртуальних серверів, який надає можливість налаштування обчислювальних ресурсів відповідно до потреб проєкту. В системі управління задачами EC2 екземпляри використовуються для розміщення додаткових компонентів, які вимагають тривалого часу роботи, таких як сервіси обробки даних, аналітичні модулі та компоненти інтеграції з зовнішніми системами. Завдяки можливості вибору різних типів екземплярів, налаштування автомасштабування та балансування навантаження, EC2 забезпечує оптимальне співвідношення ціни та продуктивності для проєкту [15].

Amazon RDS (PostgreSQL) – це сервіс реляційних баз даних, який спрощує налаштування, експлуатацію та масштабування розгортань PostgreSQL у хмарі. Для системи управління задачами RDS PostgreSQL використовується як основне сховище даних, забезпечуючи надійне збереження інформації про користувачів, проєкти, задачі та їх статуси. Сервіс автоматично виконує резервне копіювання, оновлення програмного забезпечення та моніторинг, що значно спрощує адміністрування та дозволяє зосередитись на розробці бізнес-логіки. Завдяки потенційній можливості створення читальних реплік та налаштування Multi-AZ розгортання, RDS PostgreSQL забезпечує високу доступність та продуктивність системи управління задачами при збільшенні навантаження [16].

AWS Secrets Manager – це сервіс, який надає інструментарій для захисту доступу до ресурсів та даних за допомогою шифрування та керування секретами (secrets), такими як паролі, ключі API та інші конфіденційні дані. В системі управління задачами Secrets Manager використовується для безпечного зберігання та ротації облікових даних для доступу до бази даних, зовнішніх API та інших сервісів. Інтеграція Secrets Manager з Lambda та іншими компонентами дозволяє автоматизувати процес отримання та оновлення секретів, уникаючи зберігання конфіденційної інформації в коді та конфігураційних файлах [17].

Amazon API Gateway – це керований сервіс для створення, публікації, підтримки, моніторингу та захисту REST, HTTP та WebSocket API. У системі управління задачами API Gateway виконує роль єдиної точки входу для всіх клієнтських запитів, забезпечуючи маршрутизацію до відповідних Lambda-функцій та інших бекенд-сервісів. Завдяки вбудованим механізмам авторизації, контролю доступу та обмеження швидкості запитів, API Gateway забезпечує безпеку та стабільність API, а інтеграція з AWS CloudWatch дозволяє проводити моніторинг продуктивності та виявляти потенційні проблеми [17].

Таким чином, інтеграція сервісів AWS, зокрема Lambda, VPC, EC2, RDS PostgreSQL, Secrets Manager та API Gateway, забезпечила створення надійної,

безпечної та масштабованої серверної архітектури системи управління задачами при оптимальному співвідношенні продуктивності та вартості.

1.4 Висновки до розділу 1

У першому розділі було проведено аналіз систем управління завданнями та технологій для їх розробки. Виконано детальний огляд аналогів, таких як Wrike, JIRA та OpenProj, з визначенням їхніх переваг та недоліків, що дозволило сформувати розуміння функціональних можливостей сучасних систем даного типу. Обґрунтовано вибір Angular як технології для фронтенд-розробки з урахуванням його компонентно-орієнтованої архітектури, системи впровадження залежностей, TypeScript-інтеграції та інших переваг, що забезпечують необхідну структурованість, безпеку та масштабованість системи. Також досліджено можливості хмарних сервісів AWS, зокрема Lambda, VPC, EC2, RDS PostgreSQL, Secrets Manager та API Gateway, для реалізації серверної частини системи, що дозволило створити надійну, безпечну та економічно ефективну серверну архітектуру при оптимальному співвідношенні продуктивності та вартості.

2 ПРОЄКТУВАННЯ СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ

2.1 Аналіз функціональних та нефункціональних вимог до системи

Перед початком розробки програмного продукту, виконавши аналіз переваг та недоліків конкурентів (аналогів розроблюваного програмного продукту) слід провести детальний аналіз вимог до системи, що дозволить сформувати розуміння майбутнього продукту, визначити його межі та можливості, а також встановити критерії успішності проєкту. Такий підхід забезпечує відповідність кінцевого програмного рішення потребам користувачів, мінімізує ризики невдачі проєкту та створює фундамент для подальших етапів проєктування та розробки.

Функціональні вимоги визначають конкретні функції та можливості, які повинен виконувати фінальний програмний продукт, описуючи його поведінку за різних умов використання. Коротко список функціональних вимог виглядає наступним чином:

- створення, редагування та видалення завдань із можливістю встановлення пріоритетів;
- категоризація завдань за проєктами та типами;
- призначення завдань конкретним виконавцям із повідомленнями про зміни;
- встановлення термінів виконання із системою нагадувань та сповіщень;
- відстеження прогресу виконання завдань у режимі реального часу;
- коментування завдань та обмін файлами між учасниками у режимі чату (повідомлень);
- генерування звітів про виконану роботу з різними фільтрами;
- інтеграція з календарем для синхронізації термінів;
- механізми контролю доступу та розмежування прав користувачів;
- пошук та фільтрація завдань за різними параметрами.

Система управління завданнями повинна забезпечувати повний життєвий цикл завдань від створення до завершення, підтримуючи всі необхідні функції для ефективної командної роботи. Особливу увагу слід приділити інтуїтивному інтерфейсу створення та редагування завдань, який дозволяє швидко вносити зміни та встановлювати різні атрибути (пріоритет, категорія, статус). Важливим аспектом є наявність системи сповіщень. Механізми відстеження прогресу мають відображати як загальний стан проєкту, так і деталізовану інформацію щодо окремих завдань із можливістю візуалізації даних у kanban-дошці або списку.

Нефункціональні вимоги визначають якісні характеристики системи, які не пов’язані безпосередньо з її функціональністю, але є критичними для забезпечення належної продуктивності, надійності та зручності використання програмного продукту. Категоріально нефункціональні вимоги до системи наступні:

- продуктивність – система повинна забезпечувати швидкодію при одночасній роботі 20 користувачів;
- надійність – доступність системи повинна становити не менше 99.5% часу;
- масштабованість – коректна обробка зростаючої кількості даних;
- безпека – захист даних через шифрування з потенційним підключенням двофакторної автентифікації користувачів у майбутніх оновленнях системи;
- зручність використання – інтуїтивно зрозумілий інтерфейс, який не потребує спеціального навчання;
- адаптивність – коректне відображення на різних пристроях (комп’ютери, планшети та мобільні телефони);
- локалізація – підтримка кількох мов інтерфейсу;
- відновлюваність – можливість відновлення даних після збоїв.

Забезпечення високої продуктивності та надійності системи є ключовими факторами її успішного впровадження, особливо в умовах

зростаючого навантаження та збільшення кількості користувачів. Система повинна забезпечувати час відгуку не більше 2 секунд для 95% операцій навіть при піковому навантаженні, а також підтримувати безперебійну роботу з мінімальними плановими простоями для технічного обслуговування. Архітектура системи має передбачати горизонтальне масштабування для адаптації до зростаючих потреб бізнесу без необхідності повного перепроєктування.

Особлива увага має бути приділена питанням безпеки, включаючи захист від основних видів кібератак, регулярне резервне копіювання даних та аудит дій користувачів. Інтерфейс користувача повинен відповідати сучасним принципам UX/UI дизайну, забезпечуючи однаково комфортну роботу на різних пристроях та для користувачів з різним рівнем технічної підготовки.

Отже, виконавши аналіз функціональних та нефункціональних вимог, було отримано основу для подальшого детального проєктування та розробки програмного продукту.

2.2 Концептуальне проєктування інтерфейсу та варіантів використання

Концептуальне проєктування є важливим етапом розробки програмного забезпечення, що дозволяє визначити основні принципи взаємодії користувачів із системою.

Даний рівень проєктування значно знижує ризики створення системи, яка не відповідає очікуванням користувачів, та забезпечує орієнтир для подальших етапів розробки.

На рисунку 2.1 наведено діаграму варіантів використання (прецедентів) системи управління завданнями, яка відображає ключові функціональні можливості та взаємодію різних типів користувачів із системою.

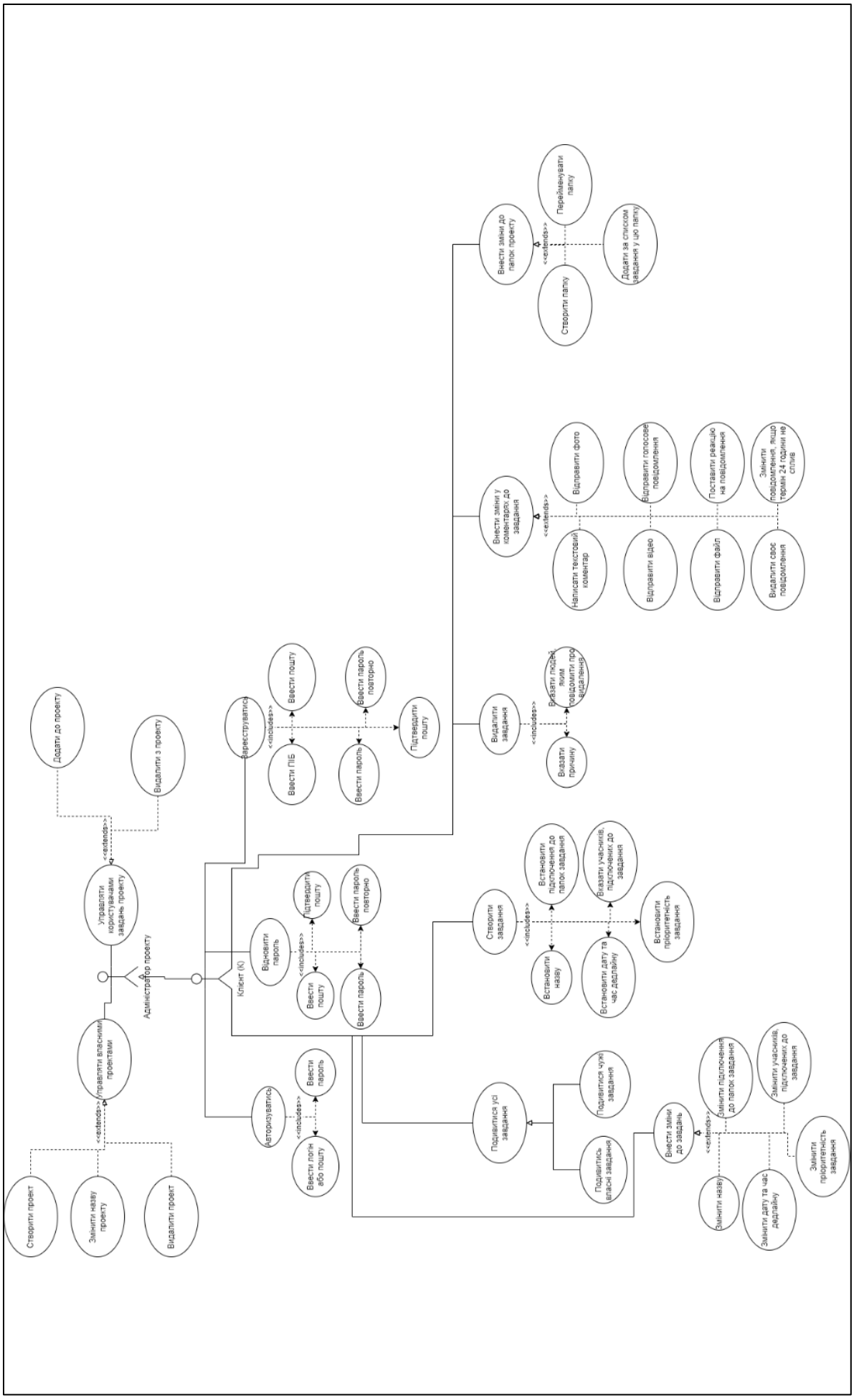


Рисунок 2.1 – Діаграма прецедентів системи управління завданнями

Діаграма прецедентів демонструє взаємодію двох основних типів користувачів із системою: адміністратора проєкту та користувача (який може бути авторизованим чи неавторизованим). Адміністратор проєкту має розширені повноваження, включаючи створення, зміну назви, видалення проєктів та управління користувацькими завданнями проєкту. Звичайний користувач після авторизації отримує можливість працювати із завданнями: створювати їх, видаляти, встановлювати параметри (назву, пріоритетність, дату та час дедлайну), та організовувати завдання в папки.

Система передбачає процеси автентифікації та авторизації з такими варіантами використання як введення логіну/пошти, пароля, його підтвердження та відновлення. Діаграма відображає також комунікаційний аспект системи через функції додавання коментарів різних типів: текстових, фото, відео та файлових вкладень.

Діаграма використовує два типи структурних відношень для демонстрації взаємозв’язків між різними варіантами використання: “includes” та “extends”. Відношення типу “includes” позначає обов’язкове включення одного варіанту використання в інший. Наприклад, процес реєстрації користувача обов’язково включає введення паролю та його підтвердження, а створення завдання включає встановлення назви. Відношення типу “extends”, натомість, позначає опціональне розширення базового варіанту використання за певних умов. Так, управління користувацькими завданнями проєкту розширюється можливостями додавання та видалення цих завдань, а введення змін до завдань розширюється функціями зміни назви, пріоритетності, підключення до папок та встановлення дати й часу дедлайну.

При проєктуванні програмного забезпечення критичну роль відіграє прототипування (створення макетів) головних компонентів фінального програмного продукту. Для розробки системи управління завданнями було спроєктовано три макети інтерфейсу – стартова сторінка, головний екран (kanban-дошка обраного користувачем проєкту) та коментарі до завдання у вигляді чату. На рисунку 2.2 наведено макет вікна привітання користувача при вході на вебсайт.

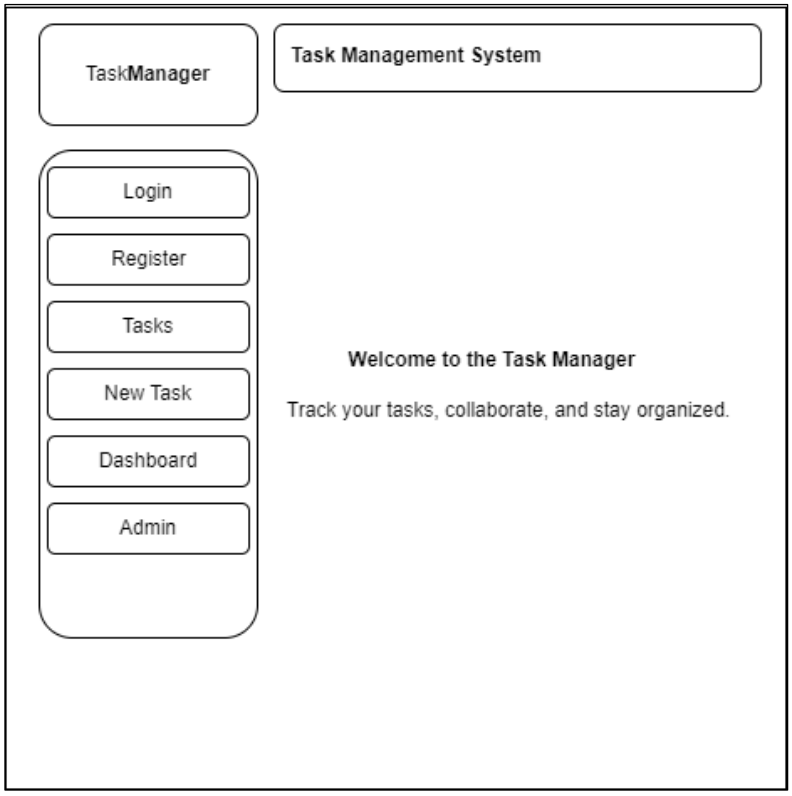


Рисунок 2.2 – Макет стартової сторінки

Відповідно до наведеного макету, користувача зустрічає повідомлення-привітання та три блоки з інформацією. Лівий верхній блок – блок лого системи управління завданнями (слово “TaskManager”, де слово “Task” – буде пофарбовано у білий колір, а слово “Manager” – у синій).

Лівий центральний блок – навігаційний блок з шістьома кнопками (авторизація, реєстрація, список завдань, створити нове завдання, kanban-дошка та адміністратор відповідно зверху вниз).

Основним компонентом системи управління завданнями є kanban-дошка, макет якої наведено на рисунку 2.3.

Компонентами інтерфейсу головної сторінки системи є дошка, назва обраного проєкту (який є випадним списком, де елементами є проєкти, на які підключено авторизованого поточного користувача), кнопка фільтр (фільтрація демонстрації завдань у дошці), базові елементи сторінки (блок-назва системи, блок повної назви системи та, за замовчуванням, згорнутий блок навігації, який можна розгорнути для вигляду, що було наведено на рисунку 2.2). Компонент-дошка складається з блоків етапів проєкту, які на

початковому етапі розробки програмного застосунку матимуть визначені системою назви та кількість. У блоці кожного етапу проєкту, за замовчуванням, будуть розміщені завдання, сортовані за датами дедлайнів).

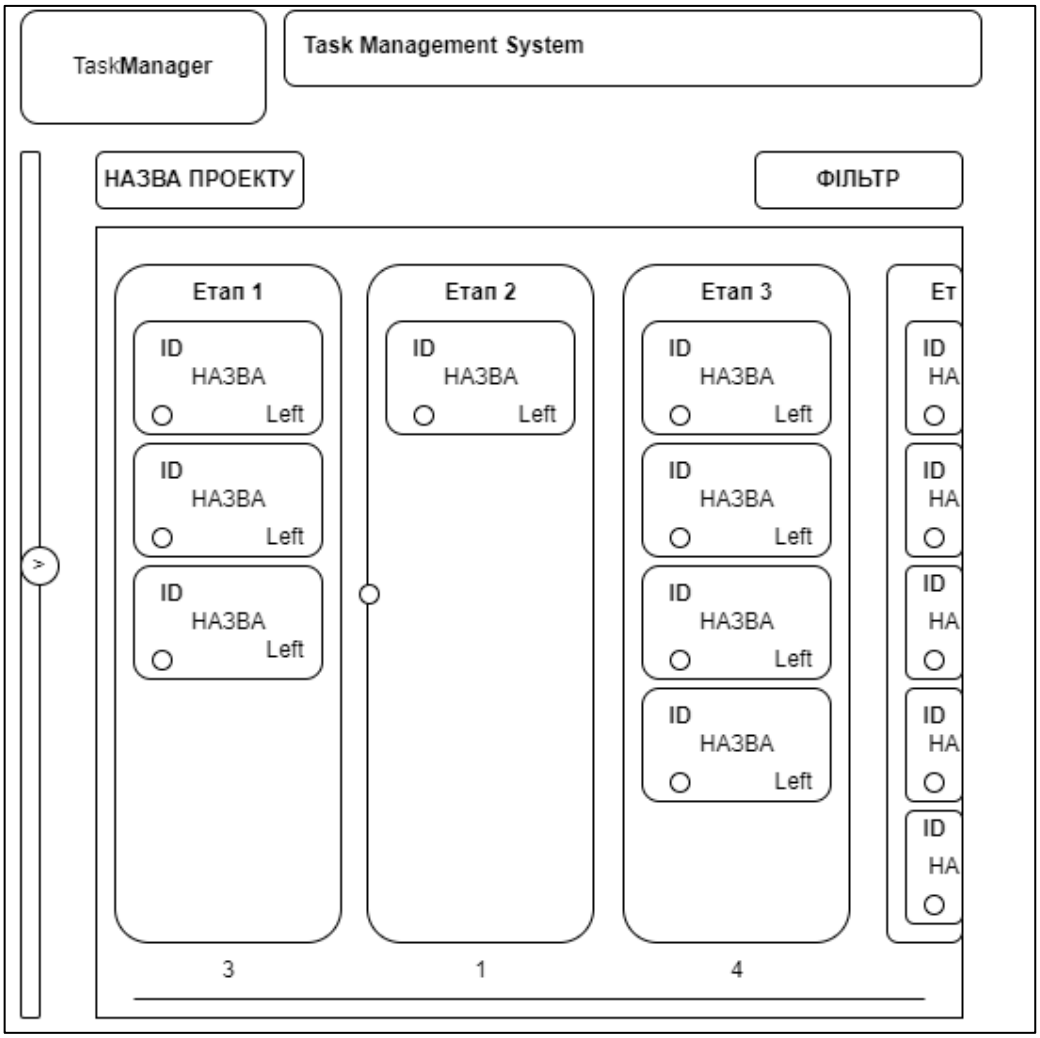


Рисунок 2.3 – Макет дошки завдань системи управління завданнями

Кожне завдання представлено у вигляді міні-блоків, де кожний блок за структурою ідентичний один до одного. Блок завдання містить ідентифікатор завдання (ID), назву, маркер пріоритетності (змінюватиме колір від синього до червоного відповідно до критичності встановленого пріоритету) та кількість часу, що залишилось до встановленого дедлайну. У разі протермінування виконання завдання, маркер пріоритетності встановлюватиме автоматично червоний колір, та нижній правий кут блоку завдання матиме червоний колір із зазначенням кількості простроченого часу з дедлайну певного завдання.

Кожне завдання проєкту матиме власні коментарі, які авторизовані користувачі зможуть залишати. На рисунку 2.4 наведено макет чату (коментарів) до обраного завдання.

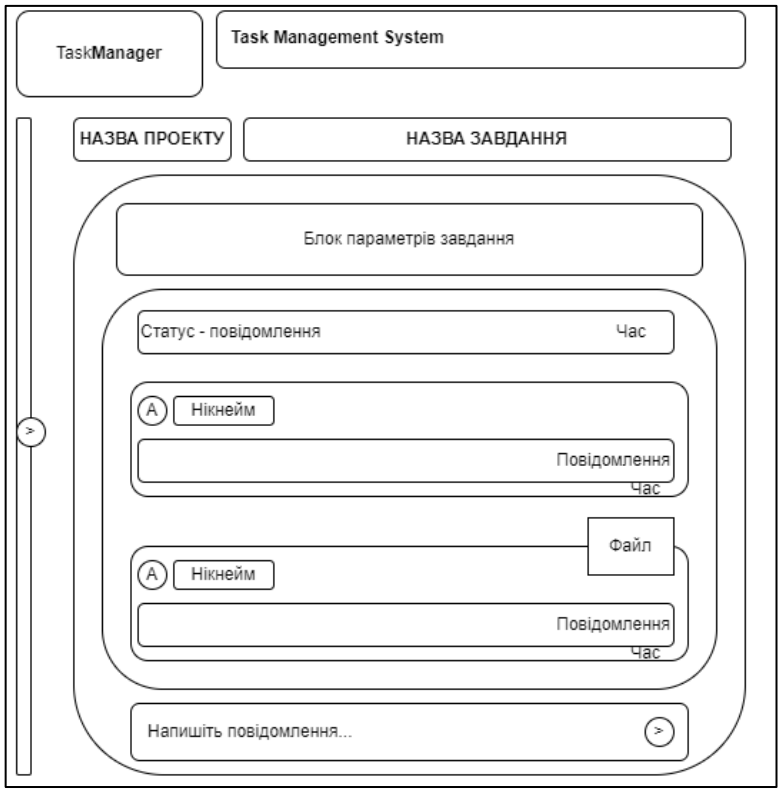


Рисунок 2.4 – Макет коментарів до обраного завдання проєкту

Відповідно до рисунку 2.4, блоки логотипу, повної назви системи та згорнутого навігаційного меню залишаються незмінними. Блок назви обраного проєкту залишається незмінним, окрім відсутності можливості змінити проєкт для перегляду. Правіше від назви проєкту було додано блок-назву завдання. По центру було розташовано блок коментарів завдання, який, по принципу роботи стандартних чатів, надаватиме огляд будь-яких змін у обраному завданні та надісланих повідомлень. Повідомлення від користувачів міститимуть інформацію про аватар користувача та нікнейм (лівий верхній кут), блок-повідомлення по центру та інформацію про час публікації у правому нижньому куту. При наявності відправленого файлу у повідомленні, файл(и) буде прикріплено у правому верхньому блоці відповідного повідомлення.

На основі проведеного концептуального проєктування системи управління завданнями можна зробити висновок, що розроблені діаграми прецедентів та макети інтерфейсу забезпечують базове розуміння функціональних можливостей майбутньої системи та шляхів взаємодії користувачів із нею.

Спроєктована система передбачає два типи користувачів з відповідним розмежуванням прав доступу, набір функцій для управління проєктами та завданнями, включаючи створення, редагування, встановлення параметрів та видалення, а також забезпечує необхідні комунікаційні можливості через систему коментарів різних типів.

Представлені макети інтерфейсу стартової сторінки, kanban-дошки та системи коментарів – відображають логічну структуру системи та зручний для користувачів дизайн, що в сукупності створює основу для подальшої розробки, мінімізуючи ризики невідповідності майбутнього програмного продукту очікуванням користувачів.

2.3 Логічне проєктування архітектури системи та структури даних

Логічне проєктування в розробці програмного забезпечення – це процес створення моделі системи, що визначає її функціональну структуру та організацію даних незалежно від технологій реалізації, зосереджуючись на бізнес-логіці та взаємодії компонентів.

На рисунку 2.5 наведено діаграму класів фронтенд-частини проєкту. Діаграма класів системи управління завданнями демонструє комплексну систему взаємодії між різними компонентами, сервісами та моделями даних.

Особливо важливими є асоціативні зв’язки між моделями даних, які відображають основні бізнес-правила системи. Так, класи Task і Project пов’язані відношенням «багато до одного», що означає приналежність кількох завдань до одного проєкту.

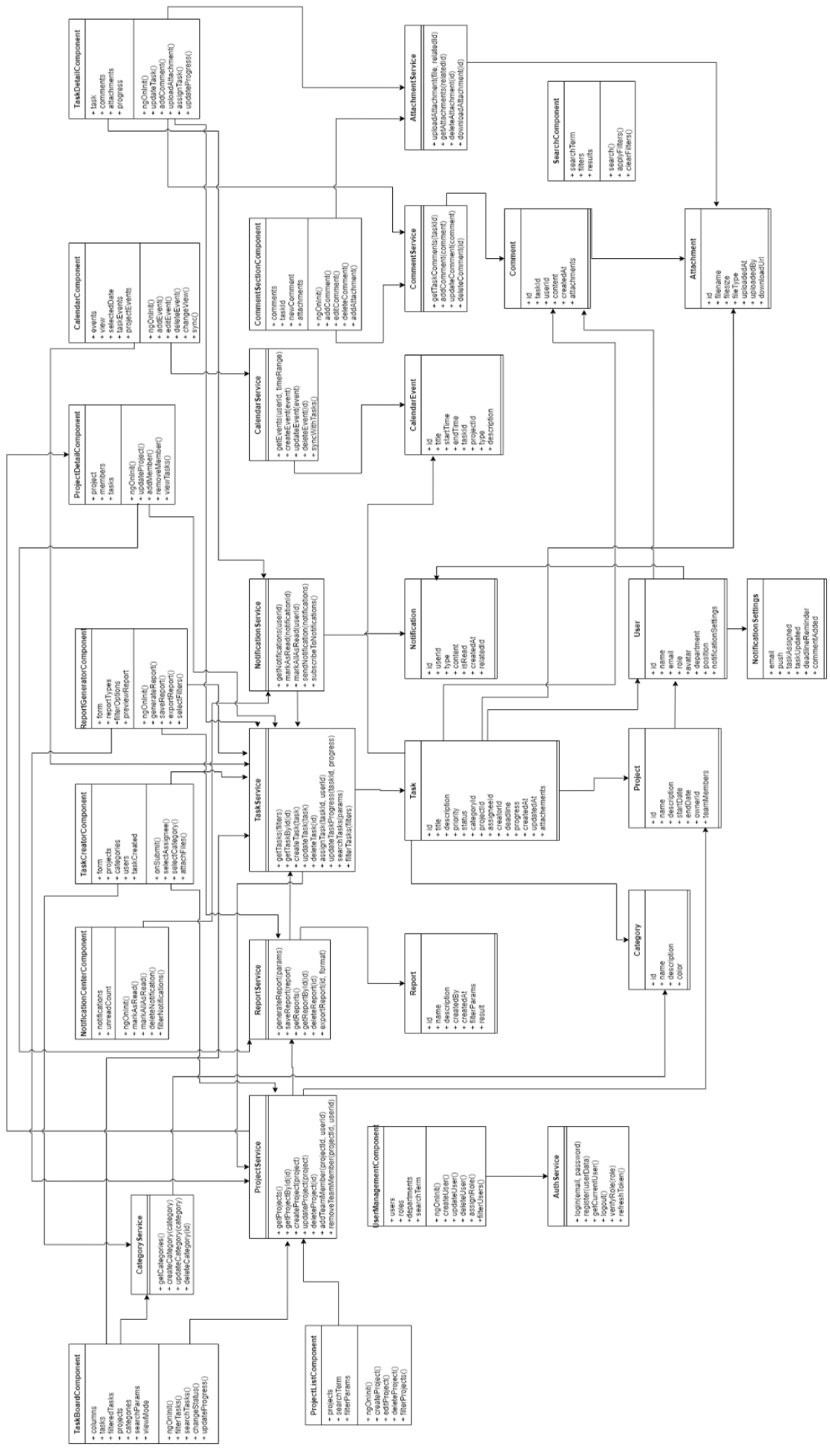


Рисунок 2.5 – Діаграма класів фронтенд-частини системи управління завданнями

Між Task і User встановлено зв’язок типу «багато до багатьох» через поле assignedTo, що дозволяє призначати завдання багатьом користувачам. Також, між Project і User існує зв’язок «багато до багатьох» через колекцію teamMembers, що дає можливість одному користувачу бути учасником кількох проєктів, а проєкту мати команду з кількох користувачів.

Детальніше розглядаючи зв’язки з коментарями, наведено, що клас Comment має асоціацію «багато до одного» з класом Task через поле taskId, а також з класом User через поле authorId. Така структура дозволяє кожному завданню мати множину коментарів, а кожен коментар пов’язаний з конкретним автором.

Крім того, між класами Comment і Attachment існує зв’язок «один до багатьох», що забезпечує можливість прикріплення файлів до коментарів. Це створює ієрархічну структуру: завдання → коментарі → вкладення, що відповідає функціональним вимогам системи щодо комунікації та обміну файлами між учасниками.

Сервіси в системі мають залежності від відповідних компонентів та моделей даних. Наприклад, TaskService використовується TaskBoardComponent, TaskDetailComponent та TaskCreatorComponent, що демонструє принцип впровадження залежностей в архітектурі Angular. При цьому сервіси також взаємодіють між собою: NotificationService викликається з TaskService при зміні статусу завдання для надсилання сповіщень відповідальним особам, а ReportService використовує дані з ProjectService і TaskService для генерації звітів. Така організація забезпечує гнучкість системи та можливість її розширення.

На діаграмі також відображені зв’язки категоризації: клас Category має асоціацію «один до багатьох» з класом Task, що дозволяє групувати завдання за категоріями. Клас CalendarEvent пов’язаний з Task, що забезпечує інтеграцію з календарем та відображення термінів виконання завдань. NotificationSettings має зв’язок «один до одного» з класом User, що дозволяє налаштувати індивідуальні параметри сповіщень для кожного користувача. Ці

зв’язки реалізують нефункціональні вимоги системи щодо гнучкості та адаптивності інтерфейсу до потреб конкретних користувачів.

Архітектура системи також демонструє зв’язки між компонентами інтерфейсу та сервісами через механізм впровадження залежностей Angular. Компоненти не взаємодіють безпосередньо з моделями даних, а роблять це через відповідні сервіси, що забезпечує дотримання принципу розділення відповідальності та полегшує тестування. Наприклад, ProjectDetailComponent використовує ProjectService для отримання деталей проєкту та управління його членами, а також TaskService для операцій із завданнями проєкту. Такий підхід до організації зв’язків забезпечує модульність системи та можливість її масштабування.

На рисунку 2.6 наведено IDEF1X діаграму системи управління завданнями. Представлена IDEF1X діаграма відображає логічну структуру бази даних системи управління завданнями, що складається з одинадцяти основних сутностей. Центральною сутністю є Tasks, яка містить ключові атрибути завдань: task_id, name, description, start_date, due_date, та зовнішні ключі для зв’язків з іншими сутностями.

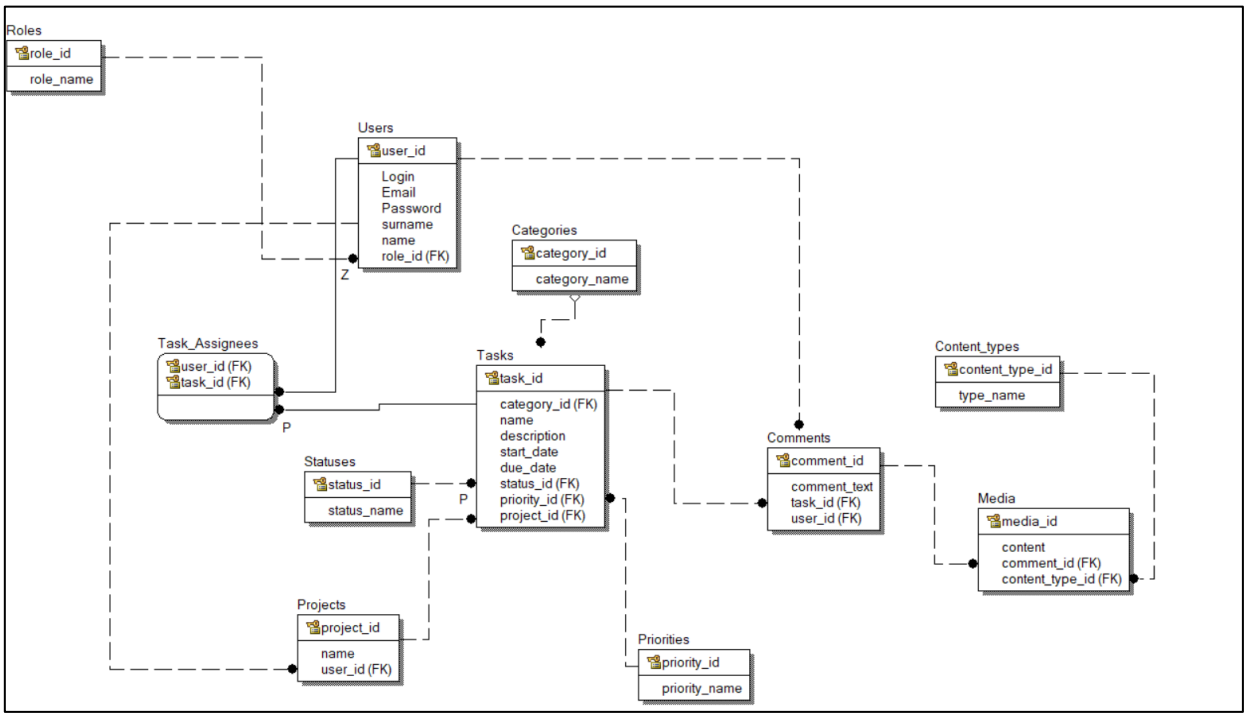


Рисунок 2.6 – IDEF1X діаграма системи управління завданнями

Завдання пов’язані з проєктами (Projects) через project_id, мають категорії (Categories), статуси (Statuses) та пріоритети (Priorities). Система підтримує управління користувачами через сутність Users з атрибутами user_id, login, email, password, surname та name, при цьому користувачі мають ролі, визначені в таблиці Roles. Важливим елементом діаграми є проміжна сутність Task_Assignees, що перетворює зв’язок «багато до багатьох» між завданнями та їх виконавцями на «один до багатьох», дозволяючи призначати кілька користувачів до одного завдання і навпаки.

Функція обговорення завдань реалізована через сутності Comments та Media. Коментарі пов’язані із завданнями через task_id та містять інформацію про автора (user_id) та текст коментаря. До коментарів можуть бути прикріплені медіафайли, що зберігаються в сутності Media з атрибутами media_id, content та зовнішніми ключами до коментаря (comment_id) та типу контенту (content_type_id). Типи контенту визначені в таблиці Content_types.

Діаграма (див. рис. 2.6) демонструє ієрархічну організацію даних: проєкти містять завдання, до яких можуть бути додані коментарі з медіафайлами. Така структура забезпечує підтримку всіх функціональних вимог системи управління завданнями, включно з категоризацією, призначенням виконавців, відстеженням прогресу та обміном файлами між учасниками проєкту.

На рисунку 2.7 наведено діаграму послідовностей прецеденту «Отримання коду підтвердження пошти користувачем при реєстрації».

Процес отримання коду підтвердження електронної пошти починається з введення користувачем email адреси через Angular фронтенд (сторінка сайту).

Запит проходить послідовну обробку через систему безпеки AWS WAF, яка фільтрує потенційно небезпечні запити, і передається до API Gateway.

Шлюз API активує спеціалізовану Lambda-функцію верифікації електронної пошти, яка реєструє спробу в Cloudwatch для моніторингу і перевіряє в DynamoDB наявність вказаної адреси.

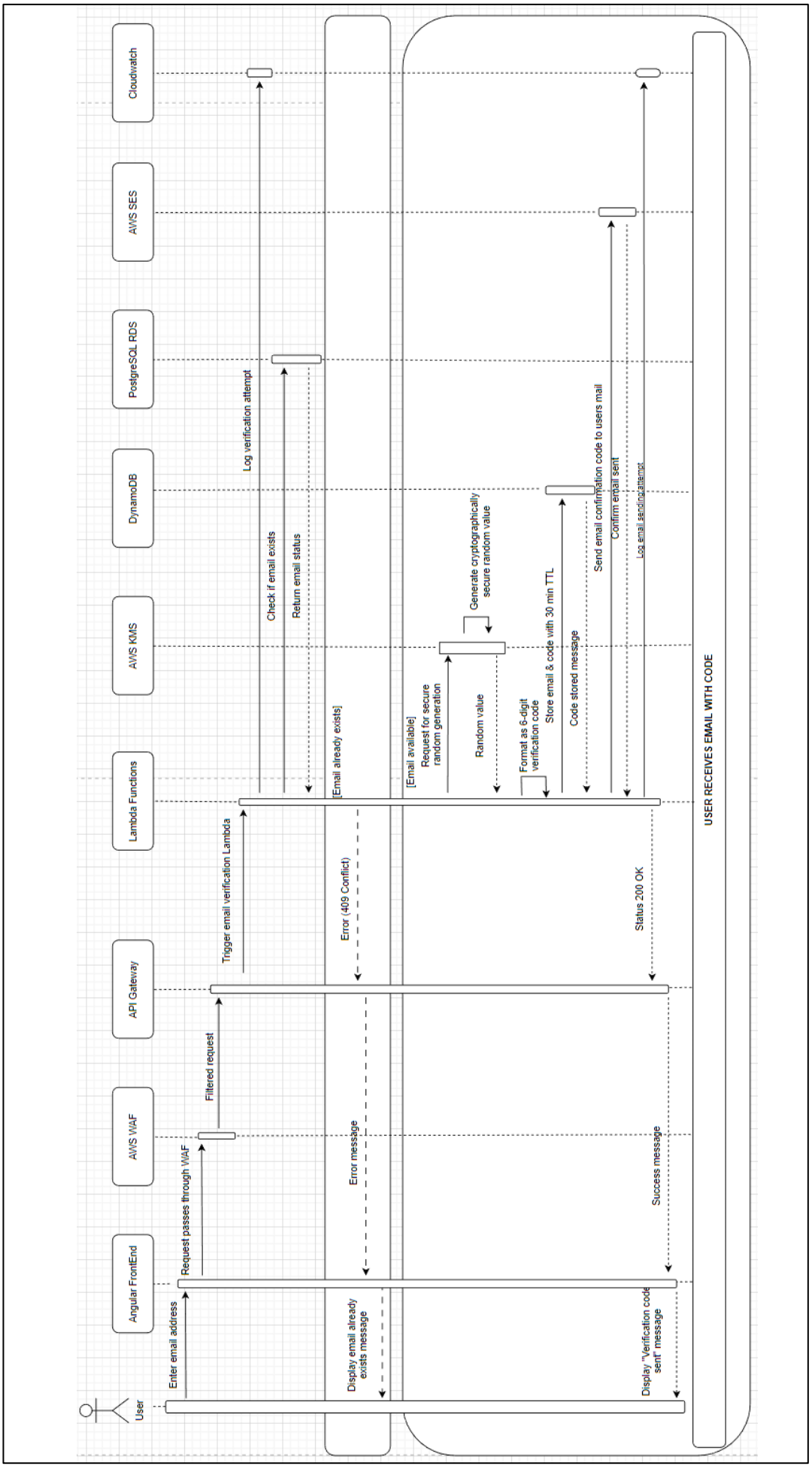


Рисунок 2.7 – Діаграма послідовностей прецеденту «Отримання коду підтвердження пошти користувачем при реєстрації»

При виявленні дублювання система повертає помилку конфлікту (409), що відображається користувачу як повідомлення про те, що email уже зареєстрований, перериваючи процес реєстрації.

Якщо ж адреса електронної пошти доступна для реєстрації, Lambda-функція запитує у сервісу AWS KMS генерацію криптографічно захищеного випадкового значення, перетворює його на 6-значний код верифікації та зберігає в DynamoDB з обмеженим терміном дії (30 хвилин). Після успішного збереження Lambda використовує AWS SES для надсилання коду підтвердження на вказану адресу, фіксуючи спробу відправки в Cloudwatch, а API Gateway повертає статус успішного виконання (200 OK). Фронтенд відображає користувачу повідомлення про успішне надсилання коду, а користувач отримує електронний лист з кодом верифікації, який використовуватиме для підтвердження володіння адресою на наступному етапі реєстрації.

Тепер розглянемо прецедент, коли користувач вводить код та після валідації введеного коду підтвердження пошти заповнює форму. На рисунку 2.8 наведено діаграму послідовностей другого етапу реєстрації користувача.

Процес підтвердження електронної пошти починається з введення користувачем отриманого верифікаційного коду через інтерфейс Angular FrontEnd (сторінка сайту). Запит проходить через систему безпеки AWS WAF, яка забезпечує первинну фільтрацію зловмисних запитів, далі передається до API Gateway, який активує спеціалізовану Lambda-функцію перевірки коду.

Ця функція реєструє спробу верифікації в Cloudwatch для аудиту та звертається до DynamoDB для перевірки відповідності введеного коду збереженому значенню для даної електронної пошти.

Після успішної перевірки Lambda-функція відзначає електронну адресу як підтверджену в DynamoDB, а система повертає успішний статус (200 OK) через API Gateway на фронтенд, який демонструє користувачу форму реєстрації для введення додаткових даних.

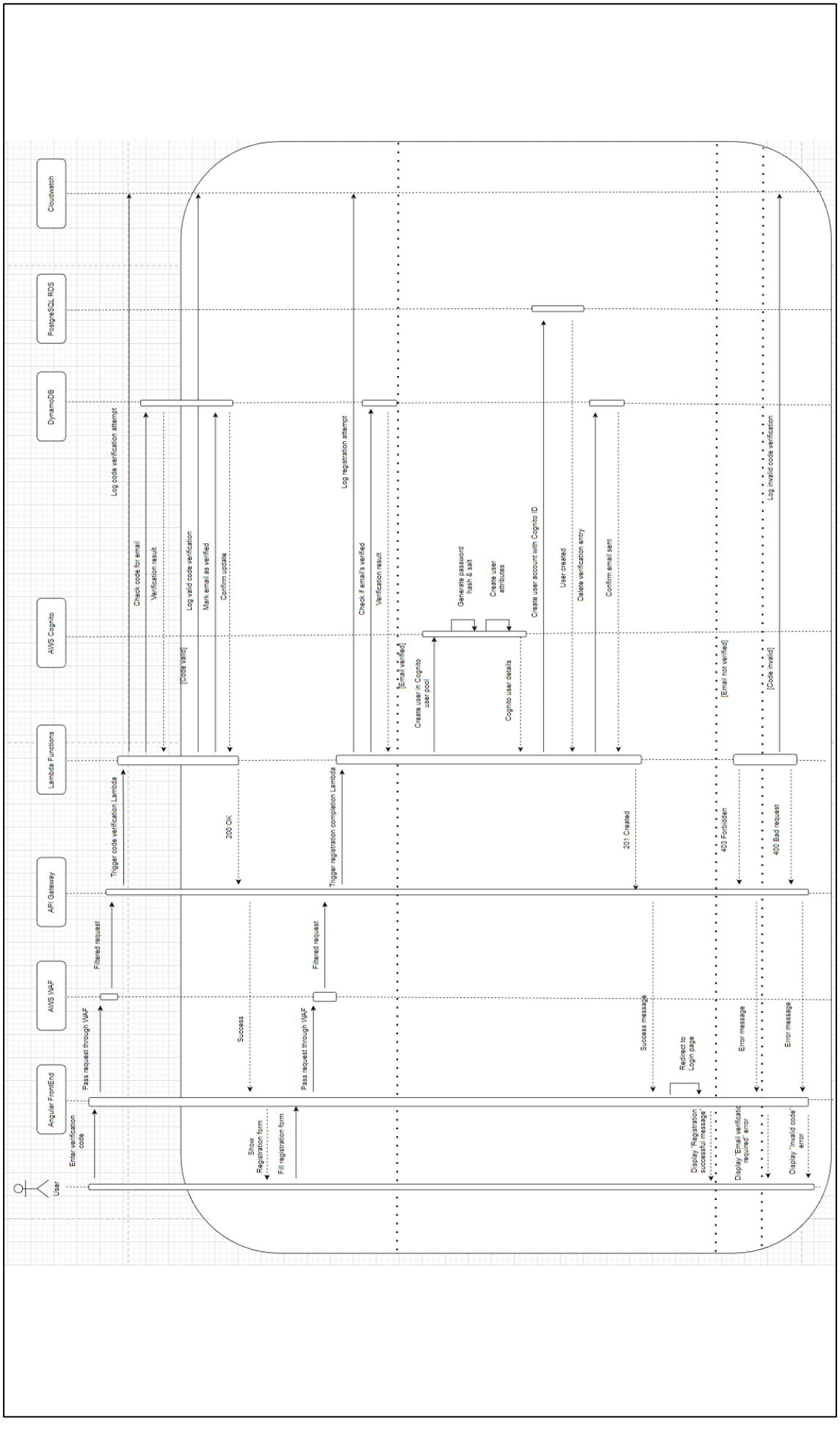


Рисунок 2.8 – Діаграма послідовностей завершення реєстрації користувача

Наступний етап процесу активується, коли користувач заповнює та надсилає реєстраційну форму з особистими даними. Ця інформація знову проходить через захисні механізми AWS WAF та API Gateway, після чого запускається Lambda-функція завершення реєстрації. Функція спочатку перевіряє в DynamoDB, чи дійсно електронна адреса була верифікована. При позитивному результаті перевірки активується процес взаємодії з AWS Cognito – сервісом управління автентифікацією та авторизацією. Lambda-функція створює користувацький пул у Cognito, генерує захищений хеш пароля та створює атрибути користувача з наданих даних.

Завершальний етап реєстрації передбачає створення повноцінного облікового запису в AWS Cognito з унікальним ідентифікатором, збереження інформації про нового користувача в PostgreSQL RDS (реляційна база даних) та видалення тимчасового запису верифікації з DynamoDB, оскільки він уже виконав свою функцію. Після успішного виконання цих операцій Lambda-функція повертає статус 201 Created через API Gateway, і фронтенд відображає користувачу повідомлення про успішну реєстрацію та автоматично перенаправляє на сторінку входу до системи. У випадку виникнення будь-яких помилок під час процесу (наприклад, недійсний код або невалідна електронна пошта), система повертає відповідні коди помилок (400 Bad Request або 403 Forbidden), а користувачу відображаються інформативні повідомлення про характер проблеми.

Наведена діаграма послідовностей детально ілюструє мікросервісну архітектуру системи та інтеграцію різних сервісів AWS для забезпечення безпечного процесу реєстрації. Використання AWS Cognito для управління обліковими записами, DynamoDB для зберігання тимчасових даних верифікації та PostgreSQL RDS для довгострокового зберігання користувацької інформації демонструє продуманий підхід до розробки масштабованої та безпечної системи автентифікації, що відповідає сучасним стандартам безпеки та вимогам до користувацького досвіду.

Отже, у розділі логічного проєктування було розроблено та детально проаналізовано комплексну архітектуру системи управління завданнями, що

включає діаграму класів фронтенд-компонентів з їх взаємозв'язками, IDEF1X діаграму структури бази даних з ключовими сутностями та діаграми послідовностей процесу автентифікації, які сукупно забезпечують надійну основу для реалізації функціональних та нефункціональних вимог системи.

2.4 Фізичне проєктування компонентів та схеми розгортання

Для демонстрації фізичної організації системи на рисунку 2.9 наведено діаграму компонентів фронтенд-частини проєкту системи управління завданнями.

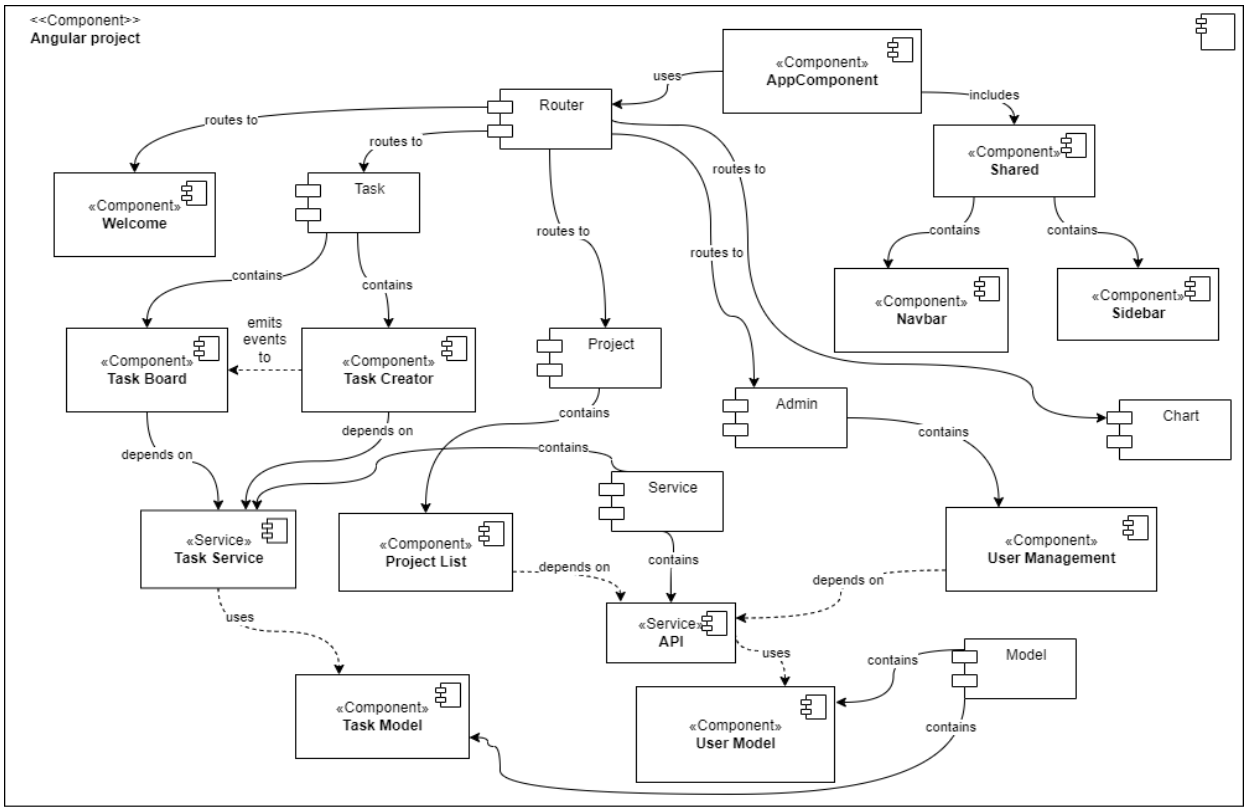


Рисунок 2.9 – Діаграма компонентів фронтенд-частини застосунку

Діаграма компонентів фронтенд-частини системи управління завданнями, розробленої на Angular, демонструє модульну компонентно-орієнтовану архітектуру, що відповідає основним вимогам, зазначеним у функціональній специфікації. Центральними елементами системи є

компоненти для роботи з проєктами (Project) та завданнями (Task), які взаємодіють через Router для навігації між різними частинами додатку. Компонент Task містить два критичні підкомпоненти: TaskBoard для візуалізації kanban-дошки та TaskCreator для створення й редагування завдань, що взаємодіють через систему подій (emits events to). Ці компоненти залежать від TaskService, який абстрагує взаємодію з бекендом через API-сервіс, що у свою чергу використовує моделі даних TaskModel та UserModel для структурованого обміну інформацією.

Архітектура включає спільні компоненти (Shared), що містять Navbar та Sidebar для навігації, адміністративний модуль (Admin) з компонентом управління користувачами (UserManagement) та компонент для візуалізації аналітичних даних (Chart). Така структурна організація забезпечує реалізацію функціональних вимог системи: створення та редагування завдань, категоризацію, призначення виконавців, встановлення термінів, відстеження прогресу та коментування. Використання сервісного шару для доступу до API дозволяє ефективно застосовувати принцип впровадження залежностей Angular, що значно спрощує тестування та підтримку коду. Взаємодія з AWS Lambda та іншими AWS-сервісами здійснюється через API-сервіс, який інкапсулює деталі комунікації з бекендом, забезпечуючи безперебійну роботу системи та відповідність нефункціональним вимогам щодо продуктивності та безпеки.

З метою демонстрації розподілу програмних компонентів по апаратних вузлах, їх взаємозв’язки та конфігурації наведено діаграму розгортання сервісів AWS бекенд-частини проєкту (див. рис. 2.10).

Представлена діаграма розгортання демонструє високорівневу архітектуру системи управління завданнями, побудовану на основі хмарних сервісів AWS. Рішення складається з клієнтської частини, що взаємодіє з серверною інфраструктурою через інтернет, проходячи через безпековий шлюз AWS WAF для захисту від зловмисних атак та брандмауер. Серверна частина розгорнута у віртуальному приватному хмарному середовищі (VPC)

AWS і розподілена між двома зонами доступності (eu-north-1 та eu-north-2) для забезпечення високої доступності та відмовостійкості.

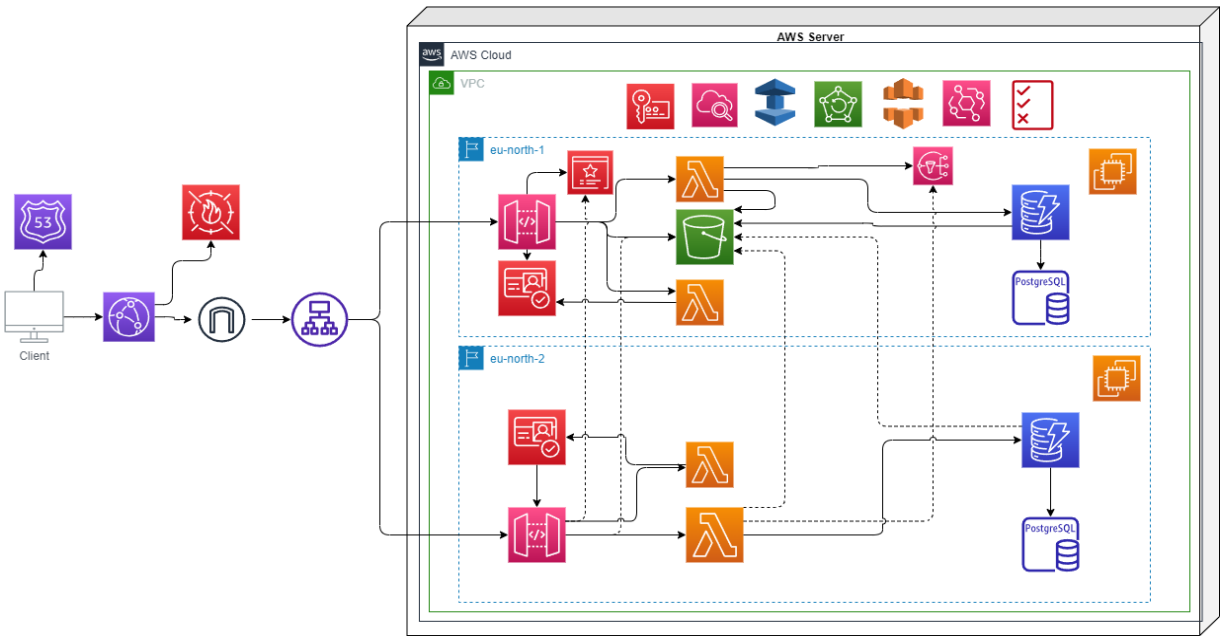


Рисунок 2.10 – Діаграма розгортання бекенд-частини проєкту

В кожній зоні розгорнуті компоненти API Gateway для маршрутизації запитів, набір Lambda-функцій для реалізації бізнес-логіки та доступу до даних, а також екземпляри DynamoDB та PostgreSQL RDS для зберігання різних типів даних системи.

Така архітектура розгортання відповідає нефункціональним вимогам системи, зазначеним у кваліфікаційній роботі: забезпечує надійність (99.5% доступності) завдяки дублюванню компонентів у різних зонах доступності; підтримує масштабованість через використання безсерверної архітектури Lambda, що автоматично адаптується до навантаження; гарантує безпеку даних через ізольоване VPC-середовище та керований доступ до ресурсів; забезпечує відновлюваність через реплікацію баз даних. Інтеграція сервісів моніторингу (позначені в правому верхньому куті діаграми) дозволяє відстежувати продуктивність системи в режимі реального часу та оперативно реагувати на інциденти, що в сукупності створює надійну основу для розгортання та експлуатації системи управління завданнями з високими вимогами до якості обслуговування.

В даному розділі було представлено технічну реалізацію системи управління завданнями через діаграму компонентів, що демонструє модульну структуру фронтенд-частини на Angular, та діаграму розгортання, яка відображає розподіл серверної інфраструктури на базі AWS-сервісів у різних зонах доступності, що в комплексі забезпечує відповідність функціональним та нефункціональним вимогам системи щодо надійності, масштабованості, безпеки та відновлюваності даних.

2.5 Висновки до розділу 2

У розділі проєктування системи управління завданнями було виконано аналіз функціональних та нефункціональних вимог, що дозволило визначити можливості системи та критерії її успішності; розроблено концептуальну модель через діаграму прецедентів та макети інтерфейсу, які візуалізують взаємодію користувачів із системою; створено логічну архітектуру за допомогою діаграми класів, IDEF1X-моделі бази даних та діаграм послідовностей для процесу реєстрації, що забезпечує структуровану організацію даних та бізнес-логіки; а також розроблено фізичну модель реалізації через діаграми компонентів і розгортання, які відображають технічну реалізацію на базі Angular та AWS-сервісів у різних зонах доступності, що в сукупності формує надійну архітектурну основу для подальшої розробки системи з дотриманням визначених функціональних та нефункціональних вимог.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Реалізація компонентів клієнтської частини на Angular

Реалізація клієнтської частини для системи управління завданнями на базі Angular включає налаштування взаємодії з серверною частиною через API, що обробляється за допомогою AWS Lambda. Основним завданням є створення інтерфейсу для додавання, редагування, видалення та перегляду завдань в реальному часі. Використання AWS Lambda дозволяє реалізувати масштабовану серверну логіку для обробки запитів без необхідності в управлінні сервером.

На рисунку 3.1 наведено компонент app-login.ts, який реалізує форму для авторизації користувача на основі фреймворку Angular.

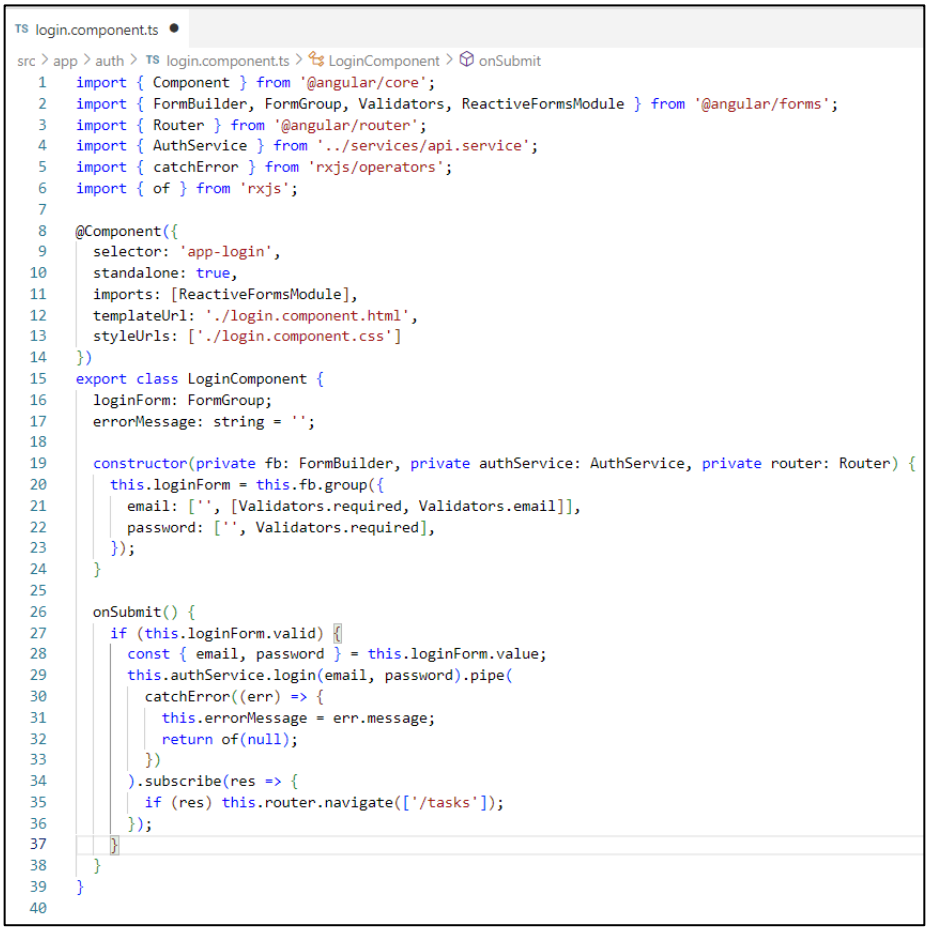


Рисунок 3.1 – Компонент app-login.ts

Компонент відповідає за обробку форми авторизації користувача в додатку. Він використовує реактивні форми (Reactive Forms) в Angular, що дозволяє створювати складні форми з різноманітними валідаціями та асинхронними запитами.

Розглянемо основні етапи роботи компонента. Перший етап – створення форми та її валідація:

– форма складається з двох полів: email та password;
– поле email має два валідатори: Validators.required (перевіряє, що поле не порожнє) та Validators.email (перевіряє, чи введено правильний формат email-адреси);
– поле password має валідатор Validators.required, який перевіряє, чи введено значення в це поле.

Реактивні форми дозволяють програмно керувати станом полів (наприклад, відображати помилки, коли значення некоректне). Розглянемо процес обробки відправки форми (onSubmit()):

– коли користувач натискає кнопку відправки форми, викликається метод onSubmit();
– якщо форма валідна (усі поля коректно заповнені), то: з форми витягується значення email та password, далі ці дані передаються в метод login() сервісу AuthService, що відповідає за взаємодію з API для авторизації користувача.

Метод login() повертає Observable, який використовується для асинхронного запиту на сервер (наприклад, перевірка правильності введених даних).

Розглянемо механізм обробки помилок:

– якщо під час запиту до API виникає помилка (наприклад, якщо введено неправильний пароль або email), це перехоплюється за допомогою оператора catchError з RxJS;
– помилка зберігається у змінній errorMessage, що дозволяє вивести її на екран для користувача;

– оскільки Observable має продовжувати виконання після обробки помилки, замість помилки повертається Observable, який містить значення null.

Якщо авторизація пройшла успішно (отримано позитивну відповідь від сервера), за допомогою Router користувача перенаправляють на сторінку /tasks (ймовірно, сторінка завдань або основного інтерфейсу додатку).

Сервіс AuthService відповідає за логіку авторизації: надсилання запиту на сервер для перевірки email та пароля, а також обробку відповіді від сервера. У цьому коді припускається, що цей сервіс реалізує метод login(), який приймає email і password. В результаті обробки запиту, на основі відповіді, компоненти можуть виконати різні дії (наприклад, перейти на іншу сторінку або відобразити помилку).

Важливою перевагою реактивних форм є можливість керувати полями, додавати динамічні перевірки та миттєво реагувати на зміни у формах. У цьому випадку, якщо введено некоректний email або пароль, Angular автоматично відображатиме повідомлення про помилки, якщо вони задані в шаблоні форми. Використання реактивних форм забезпечує зручність та гнучкість при роботі з формами, дозволяючи не тільки валідувати дані на клієнті, а й зручно обробляти асинхронні запити та помилки. Інтеграція з сервісом AuthService дозволяє централізовано обробляти логіку авторизації, а використання Router дає змогу ефективно переміщати користувача маршрутами у додатку після успішної авторизації. Цей компонент є основою для простого та зручного механізму входу в додаток, який можна легко розширювати та інтегрувати з іншими частинами системи.

На рисунку 3.2 наведено компонент RegisterComponent для реєстрації користувачів в Angular. Він використовує реактивні форми для обробки введених даних, а також взаємодіє з сервером через AuthService для реєстрації нових користувачів.

Розглянемо структуру компонента форми реєстрації користувачів. У компоненті визначено форму registerForm з трьома полями: name (ім’я

користувача – обов’язкове поле), email (адреса електронної пошти користувача – обов’язкове поле та перевірка формату email), password (пароль користувача – обов’язкове поле).

register.component.ts

src > app > auth > register.component.ts > ...

```ts
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators, ReactiveFormsModule } from '@angular/forms';
import { Router } from '@angular/router';
import { AuthService } from '../services/api.service';
import { catchError } from 'rxjs/operators';
import { of } from 'rxjs';

@Component({
  selector: 'app-register',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent {
  registerForm: FormGroup;
  errorMessage: string = '';

  constructor(private fb: FormBuilder, private authService: AuthService, private router: Router) {
    this.registerForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      password: ['', Validators.required],
    });
  }

  onSubmit() {
    if (this.registerForm.valid) {
      const { name, email, password } = this.registerForm.value;
      this.authService.register({ name, email, password }).pipe(
        catchError((err) => {
          this.errorMessage = err.message;
          return of(null);
        })
      ).subscribe(res => {
        if (res) this.router.navigate(['/login']);
      });
    }
  }
}
```

Рисунок 3.2 – Компонент RegisterComponent

Для кожного поля встановлено відповідні валідатори: Validators.required (перевіряє, чи не порожнє поле), Validators.email (перевіряє, чи введена коректна email-адреса).

Реактивні форми в Angular дозволяють зручно перевіряти стан форми і полів, а також надавати зворотній зв’язок користувачеві. Коли користувач натискає кнопку відправки форми, відбувається наступна послідовність подій:

– спочатку перевіряється, чи є форма коректною (всі поля коректно заповнені);

– якщо форма коректна, від неї отримуються значення полів name, email і password;
– ці значення передаються в метод register() сервісу AuthService, який виконує запит на сервер для реєстрації нового користувача.

AuthService.register() – це метод, який відправляє запит на сервер для створення нового користувача з переданими даними (ім’я, email, пароль). Далі відбувається процес обробки помилок:

– якщо під час реєстрації виникає помилка (наприклад, якщо email вже зареєстрований), це перехоплюється за допомогою оператора catchError з RxJS;
– помилка зберігається в змінній errorMessage, яку можна відобразити на екрані, щоб повідомити користувача про проблему;
– після обробки помилки повертається Observable, який містить значення null, що дозволяє продовжити виконання програми без зупинки.

Якщо реєстрація була успішною (тобто відповідь від сервера була успішною), викликається метод router.navigate([‘/login’]), що перенаправляє користувача на сторінку входу.

Сервіс AuthService реалізує логіку для роботи з API. Він забезпечує метод register(), який відповідає за взаємодію з сервером для реєстрації нових користувачів. Цей сервіс дозволяє централізовано обробляти авторизацію та реєстрацію в додатку.

Реактивні форми дозволяють керувати станом форми програмно, легко додавати валідатори та обробляти їх, зручно взаємодіяти з асинхронними запитами та обробляти помилки та стан форми. Кроки для реалізації:

– створення форми з необхідними полями;
– додавання валідаторів для кожного поля;
– взаємодія з сервером через AuthService для реєстрації користувача;
– обробка помилок і надання зворотного зв’язку користувачеві;
– перехід на сторінку входу після успішної реєстрації.

Цей компонент реалізує форму реєстрації для нових користувачів у вебдодатку. Він перевіряє коректність введених даних, відправляє їх на сервер для реєстрації через AuthService, обробляє помилки і перенаправляє користувача на сторінку входу після успішної реєстрації.

На рисунку 3.3 наведено компонент TaskBoardComponent, який призначено для відображення списку завдань (tasks) на дошці завдань. Компонент використовує Angular для отримання та групування завдань за їх статусами.



Рисунок 3.3 – Компонент TaskBoardComponent

Компонент виконує кілька основних функцій. По-перше, отримання даних. Компонент використовує сервіс taskService для отримання списку завдань через HTTP запит. Дані, що надходять у вигляді потоку (Observable),

містять завдання з різними статусами (наприклад, «design», «development», «testing»). По-друге, групування завдань. Після отримання даних компоненти групує завдання за їх статусами, тобто завдання, що знаходяться на етапі дизайну, потрапляють у одну категорію, завдання на етапі розробки – в іншу, і так далі. Це дозволяє організувати завдання на дошці за статусами. По-третє, відображення завдань. Груповані за статусами завдання можна відображати в інтерфейсі користувача, дозволяючи команді або користувачам чітко бачити, на якому етапі знаходиться кожне завдання.

Основна суть компонента полягає в тому, щоб ефективно організувати та відображати завдання, надаючи користувачам зручний інтерфейс для перегляду завдань за їх статусами.

На рисунку 3.4 наведено компонент TaskCreatorComponent в Angular реалізує інтерфейс для створення нового завдання.

task-creator.component.ts

src > app > tasks > task-creator.component.ts > ...

```
import { Component, EventEmitter, Output } from '@angular/core';
import { FormBuilder, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { Task } from '../model/task.model';

@Component({
  selector: 'app-task-creator',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './task-creator.component.html',
  styleUrls: ['./task-creator.component.css']
})
export class TaskCreatorComponent {
  @Output() taskCreated = new EventEmitter<Task>();
  form: FormGroup;

  constructor(private fb: FormBuilder) {
    this.form = this.fb.group({
      title: ['', Validators.required],
      priority: ['medium', Validators.required],
      status: ['design', Validators.required]
    });
  }

  onSubmit() {
    if (this.form.valid) {
      const task: Task = {
        id: `KAN-${Math.floor(Math.random() * 1000)}`,
        ...this.form.value
      };
      this.taskCreated.emit(task);
      this.form.reset({ priority: 'medium', status: 'design' });
    }
  }
}

```

Рисунок 3.4 – Компонент TaskCreatorComponent

У компоненті за допомогою FormBuilder створюється форма з трьома полями: title (назва завдання – обов’язкове поле), priority – пріоритет завдання (типовий – ‘medium’, обов’язкове поле) та status – статус завдання (типовий – ‘design’, обов’язкове поле).

Після заповнення форми та натискання кнопки «Створити» викликається метод onSubmit():

– перевіряється валідність форми;
– якщо форма валідна, створюється об’єкт типу Task (ідентифікатор генерується випадково);
– за допомогою декоратора @Output і EventEmitter це завдання передається назовні (до батьківського компонента);
– форма скидається до початкового стану, знову виставляючи типові значення для priority і status.

Компонент TaskCreatorComponent дозволяє динамічно створювати нові завдання з вибором пріоритету та статусу, та інтегрується в систему управління завданнями. Він не зберігає завдання самостійно, а лише передає їх далі – наприклад, на сервер або в список завдань в TaskBoardComponent.

У даному розділі було реалізовано клієнтську частину системи управління завданнями на базі Angular, що включає розробку компонентів для авторизації користувачів (app-login.ts), реєстрації нових користувачів (RegisterComponent), відображення та групування завдань за статусами (TaskBoardComponent) та створення нових завдань (TaskCreatorComponent) з використанням реактивних форм та взаємодії з серверною частиною через API на базі AWS Lambda.

3.2 Реалізація серверної логіки з використанням AWS Lambda

У рамках розробки системи управління завданнями було прийнято рішення реалізувати серверну логіку за допомогою сервісу AWS Lambda, що

забезпечує безсерверне (serverless) виконання коду у відповідь на певні події, без необхідності адміністрування серверів.

У системі клієнтська частина, створена за допомогою Angular, взаємодіє з бекендом через Amazon API Gateway, який маршрутизує HTTP-запити до відповідних AWS Lambda функцій. Lambda-функції виконують бізнес-логіку: створення, оновлення, видалення та отримання завдань користувачів.

Схема взаємодії виглядає наступним чином:

– користувач ініціює запит через Angular-інтерфейс;
– запит надсилається до API Gateway;
– API Gateway викликає відповідну Lambda-функцію;
– Lambda-функція обробляє запит, взаємодіє з базою даних та повертає результат назад;
– відповідь повертається до Angular-додатку та відображається користувачу.

Розглянемо декілька прикладів Lambda-функцій, які було використано в застосунку, для автоматизації або обробки певних запитів на сервері без необхідності використання постійно працюючого бекенд-сервера. На рисунку 3.5 наведено Lambda-функцію для отримання даних за бази даних PostgreSQL.

Функція виконує роль частини серверної логіки в системі керування завданнями. Її основне призначення – отримати перелік користувачів, призначених до проєктів, та повернути ці дані клієнтській частині через HTTP-відповідь.

Функція виконує такі кроки:

– отримує конфіденційні дані з AWS Secrets Manager, зокрема логін і пароль адміністратора бази даних, а також параметри підключення (хост, порт, назва бази, ім’я користувача);
– встановлює підключення до бази даних PostgreSQL, використовуючи отримані облікові дані;
– виконує SQL-запит: SELECT user_id, project_id FROM project_assignees; – для отримання всіх зв’язків між користувачами та проєктами;

– формує HTTP-відповідь у форматі JSON (якщо записи в таблиці відсутні – повертається статус 404 та повідомлення; якщо дані є – статус 200 і самі дані).

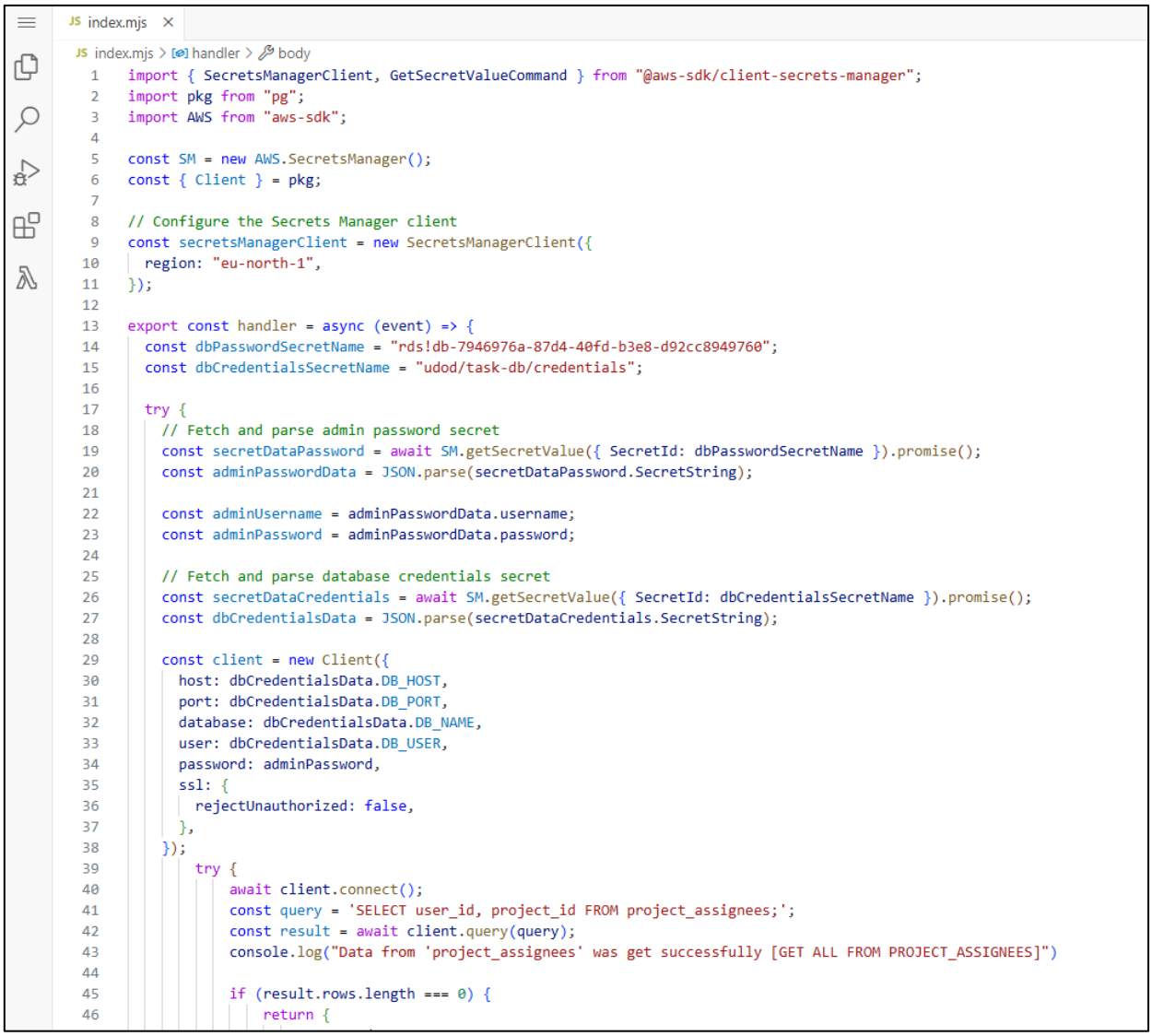


Рисунок 3.5 – Lambda-функція для отримання даних з PostgreSQL

Функція обробляє потенційні помилки, зокрема:

– помилки при підключенні до бази;

– помилки при отриманні секретів (наприклад, секрет не знайдено, доступ заборонено або запити перевищують ліміт);

– загальні помилки виконання коду.

Ця функція є частиною безсерверної архітектури (serverless) і дозволяє динамічно взаємодіяти з базою даних без постійного серверного оточення.

На рисунку 3.6 наведено Lambda-функцію для отримання переліку ролей користувачів із бази даних PostgreSQL. Вона є частиною бекенд-системи для управління завданнями. Основне її призначення – обробка запитів на отримання даних з таблиці roles (наприклад, адміністратор, розробник, менеджер тощо) та передача їх клієнтській частині застосунку у вигляді JSON-відповіді.

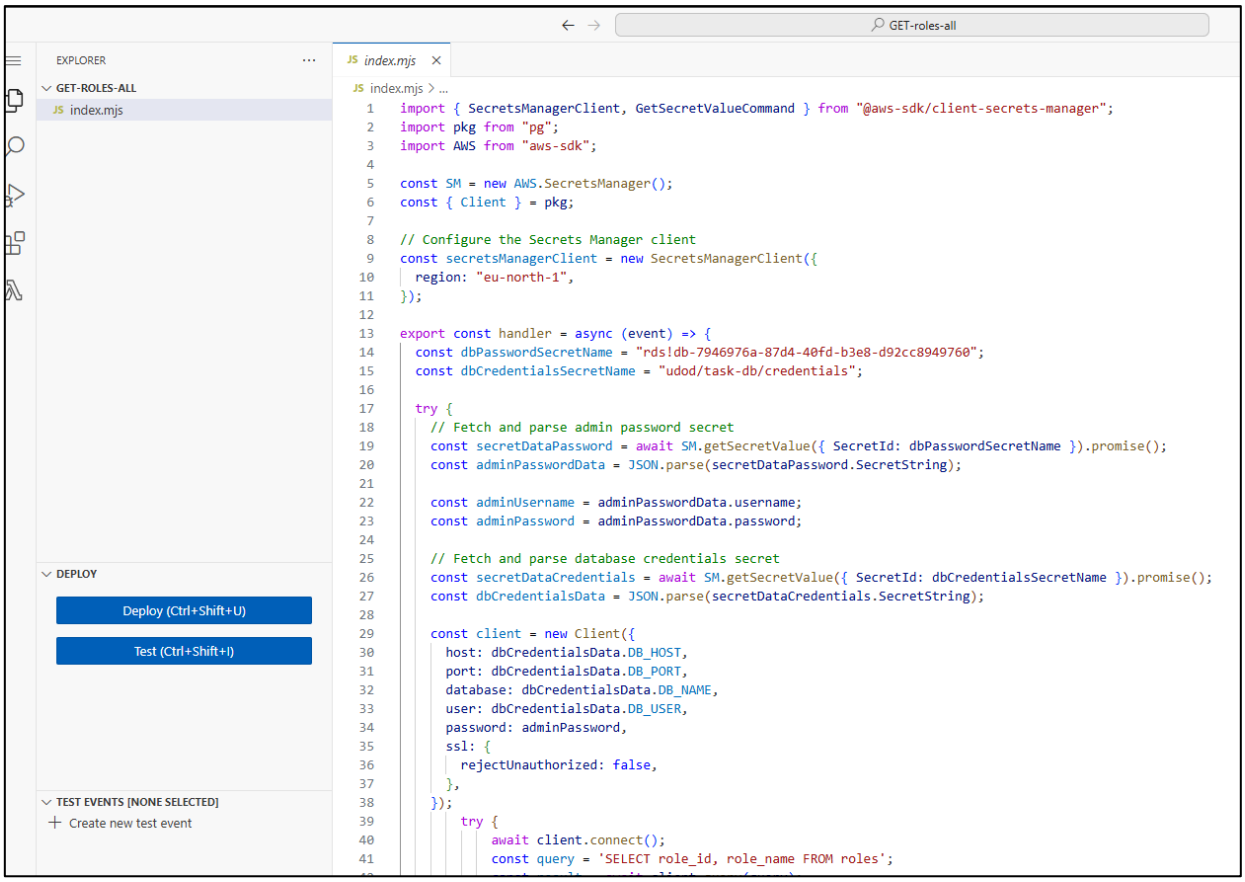


Рисунок 3.6 – Lambda-функція для отримання ролей

Функція забезпечує отримання облікових даних до бази даних за допомогою AWS Secrets Manager. Вона безпечно завантажує два критичні секрети: перший містить логін і пароль адміністратора бази, а другий включає всі необхідні параметри з’єднання з базою даних, такі як хост, порт, назва бази даних та ім’я користувача.

Після успішного отримання облікових даних функція встановлює захищене з’єднання з базою даних PostgreSQL, використовуючи отримані

параметри. Це забезпечує безпечний канал комунікації для подальшої взаємодії з базою даних. Функція виконує SQL-запит «SELECT role_id, role_name FROM roles;» для отримання всіх записів з таблиці ролей. Цей запит вибирає ідентифікатори та назви всіх ролей, наявних у системі.

Якщо таблиця ролей виявляється порожньою, функція генерує відповідь зі статусом 404 та інформативним повідомленням про відсутність даних. У випадку наявності даних у таблиці, функція формує відповідь зі статусом 200 та повертає структурований список ролей у форматі JSON. Функція включає комплексну обробку помилок. У разі виникнення проблем при взаємодії з AWS Secrets Manager, таких як відсутність секрету, відмова в доступі або перевищення ліміту запитів, генерується відповідна HTTP-відповідь з інформацією про помилку. Якщо виникають складнощі під час виконання SQL-запиту або спроби підключення до бази даних, функція повертає відповідь зі статусом 500, сигналізуючи про внутрішню помилку сервера.

Ця функція є базовим прикладом безсерверної архітектури для виконання окремих завдань у хмарному середовищі – без потреби в постійному сервері. Вона автоматично активується по запиту і завершується після виконання, що робить її ефективною для мікросервісної системи.

На рисунку 3.7 наведено Lambda-функцію, яка відповідає за створення та наповнення бази даних PostgreSQL початковими таблицями і даними для проєктного менеджменту.

Функція розпочинає свою роботу з підключення до AWS Secrets Manager для витягнення необхідних секретів. Вона отримує два критичні набори даних: пароль адміністратора бази даних та повний комплект облікових даних для підключення до бази даних PostgreSQL, який включає хост, порт, ім’я бази даних та користувача. Ці дані зберігаються в захищеному вигляді та доступні лише авторизованим сервісам. Після успішного отримання облікових даних функція використовує їх для встановлення безпечного з’єднання з базою даних PostgreSQL. Це з’єднання слугуватиме каналом для виконання всіх подальших операцій з базою даних.

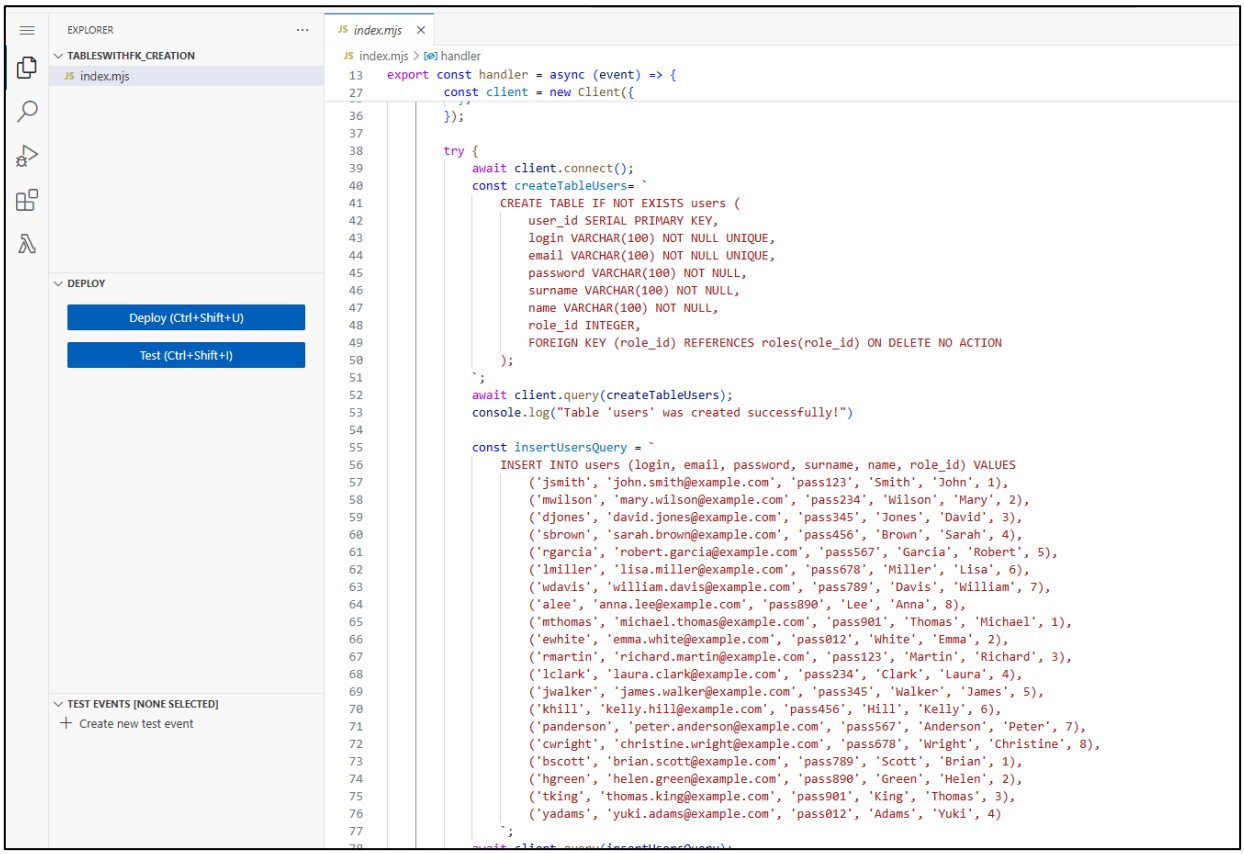


Рисунок 3.7 – Lambda-функція для створення таблиць бази даних

Функція продовжує роботу створенням структури бази даних через серію SQL-запитів. Спочатку створюється таблиця users, яка містить основну інформацію про користувачів системи та їхні ролі. Далі формується таблиця project_credentials для зберігання даних про проєкти, включаючи їхню ієрархічну структуру з батьківськими та дочірніми елементами. Таблиця project_assignees слугує для збереження зв’язків між користувачами та проєктами, над якими вони працюють. Наступною створюється таблиця tasks для зберігання детальної інформації про завдання, включаючи терміни виконання, поточні статуси, пріоритети та приналежність до конкретних проєктів. Таблиця task_assignees відповідає за фіксацію зв’язків між користувачами та призначеними їм завданнями. Для збереження комунікації між учасниками створюється таблиця comments, де зберігаються коментарі до завдань разом із датою публікації та інформацією про автора. Нарешті, таблиця media призначена для збереження різноманітних медіа файлів, пов’язаних із коментарями, таких як скріншоти, відеоматеріали чи документи.

Після успішного створення структури бази даних функція переходить до наповнення її тестовими даними. У кожну з таблиць додаються початкові записи: створюються тестові користувачі з різними ролями, проєкти різного рівня складності, завдання з різними пріоритетами та статусами, коментарі до завдань та необхідні зв’язки між усіма цими сутностями. Це дозволяє одразу тестувати функціональність системи без необхідності ручного введення даних.

Важливою частиною функції є комплексна система обробки помилок. Усі операції з базою даних обгорнуті в блоки try-catch, щоб забезпечити коректне завершення роботи навіть у випадку виникнення непередбачених ситуацій. У разі помилки на будь-якому етапі, функція генерує відповідний статус та інформативне повідомлення, яке допомагає ідентифікувати проблему. Це особливо важливо під час розгортання та налаштування системи, коли може виникнути потреба у діагностиці проблем з’єднання чи структурою бази даних.

У даному розділі було реалізовано серверну частину системи управління завданнями з використанням безсерверної архітектури на базі AWS Lambda, де взаємодія між клієнтською частиною (Angular) та бекендом відбувається через Amazon API Gateway, з розробкою функцій для отримання даних користувачів та їх ролей з PostgreSQL, а також функції для створення та наповнення бази даних необхідними таблицями (users, projects, tasks, comments тощо) та тестовими даними.

3.3 Система автентифікації та авторизації

У системі управління завданнями особливу увагу було приділено безпеці, зокрема реалізації надійної системи автентифікації та авторизації користувачів. Вона забезпечує захист доступу до функціоналу та даних відповідно до ролей та прав користувачів.

Автентифікація виконується на клієнтській стороні за допомогою Angular, який взаємодіє із серверною частиною, реалізованою через AWS

Lambda. Для автентифікації користувачів застосовується протокол JWT (JSON Web Token), що дозволяє ефективно керувати сесіями без збереження стану на сервері (stateless).

Процес автентифікації включає:

– введення користувачем логіна та пароля у вебінтерфейсі Angular;
– надсилання цих даних до AWS Lambda-функції, яка перевіряє їх у базі даних;
– у разі успішної перевірки сервер генерує JWT-токен, який повертається клієнту;
– клієнт зберігає токен (наприклад, у localStorage або sessionStorage) та додає його в заголовки наступних HTTP-запитів.

Авторизація базується на перевірці прав користувача, закодованих у JWT-токені. Кожен запит до серверу, що вимагає доступу до захищених ресурсів (наприклад, створення або редагування завдань), супроводжується передачею токена в заголовку Authorization.

AWS Lambda-функції перевіряють валідність та дійсність токена, а також рівень доступу користувача перед виконанням бізнес-логіки. У разі невідповідності або відсутності прав запит відхиляється із відповідним кодом помилки (401 або 403).

Для підвищення безпеки система може використовувати сервіси AWS, наприклад:

– AWS Cognito – для управління користувачами, їх реєстрації, автентифікації та авторизації;
– IAM ролі та політики – для контролю доступу до ресурсів AWS;
– Secrets Manager – для безпечного зберігання секретів і ключів.

Розглянемо приклади реалізації в застосунку. На рисунку 3.8 наведено компонент Lambda-функції для реалізації функціоналу авторизації та автентифікації користувача у вебзастосунку.

Функція починає свою роботу з отримання необхідних секретів з AWS Secrets Manager. У першу чергу вона завантажує пароль адміністратора бази

даних, що використовуватиметься для подальшого доступу до даних. Паралельно отримуються всі необхідні параметри для підключення до бази даних PostgreSQL, включаючи адресу сервера, порт, ім’я користувача та інші критичні дані. Також функція завантажує секретний ключ, який буде використовуватись для підпису JWT-токенів, забезпечуючи їхню безпеку та цілісність.

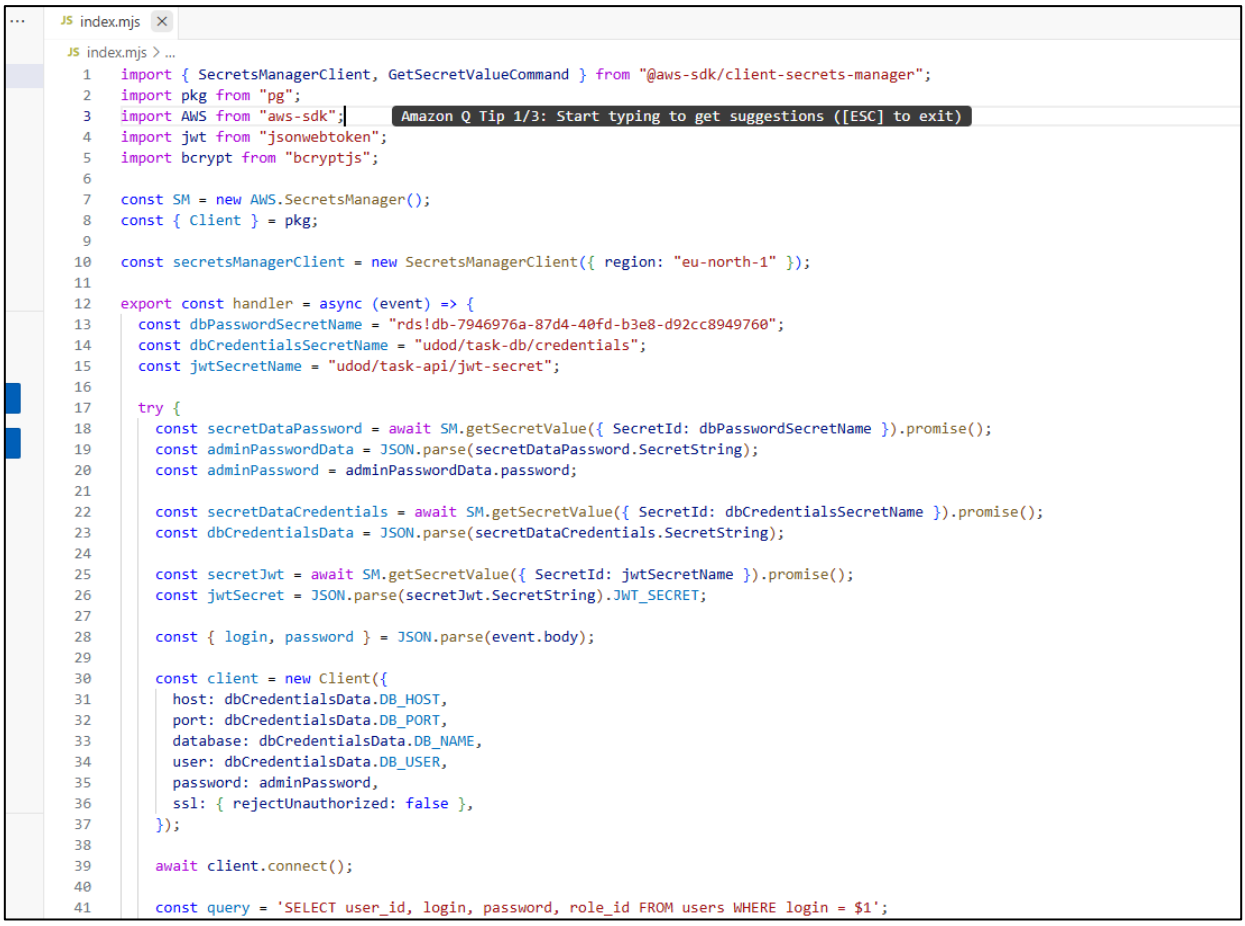


Рисунок 3.8 – Lambda-функція для авторизації і автентифікації користувачів

Після отримання всіх необхідних секретів, функція переходить до обробки вхідного запиту. Вона отримує дані логіну та пароля з тіла HTTP-запиту, які надсилаються з фронтенду користувачем. Використовуючи попередньо отримані параметри підключення, функція встановлює з’єднання з базою даних PostgreSQL. Далі вона формує та виконує SQL-запит для пошуку користувача з введеним логіном у системі, перевіряючи його наявність у базі даних.

Знайшовши відповідний запис користувача, функція переходить до етапу верифікації паролю. Вона використовує бібліотеку bcrypt для безпечного порівняння хешу паролю, збереженого в базі даних, з паролем, що був введений користувачем. Якщо введений пароль не відповідає збереженому хешу або якщо користувача з таким логіном взагалі не знайдено в системі, функція завершує роботу, повертаючи відповідь зі статусом 401, що сигналізує про помилку авторизації.

У випадку успішної перевірки паролю, функція переходить до створення JWT-токена. Вона формує токен, включаючи до нього важливу інформацію про користувача, таку як унікальний ідентифікатор, логін та ідентифікатор ролі в системі. Для забезпечення безпеки токен обмежується часом дії, зазвичай встановленим на дві години, після чого він автоматично стає недійсним і вимагає оновлення.

Завершальним етапом роботи функції є формування відповіді для клієнта. При успішній автентифікації вона повертає JSON-об’єкт, що містить згенерований токен у форматі { token: "..." }. У разі виникнення помилок на будь-якому етапі обробки запиту, функція генерує відповідь із статусом помилки (401 для проблем автентифікації або 500 для внутрішніх помилок сервера) та інформативним повідомленням про характер проблеми.

Взаємодія починається з фронтенду, розробленого на Angular, який надсилає POST-запит на API Gateway (див. рис. 3.9). Цей запит автоматично перенаправляється до описаної Lambda-функції. Користувач вводить свої облікові дані, включаючи електронну пошту або логін та пароль, які передаються для перевірки.

Після успішної автентифікації система повертає JWT-токен, який фронтенд зберігає у локальному або сесійному сховищі браузера. Для всіх наступних запитів до захищених ресурсів API цей токен додається в заголовок запиту у форматі Authorization: Bearer <token>. Система використовує цей токен для перевірки особи користувача та визначення його прав доступу до різних функцій та ресурсів.

```typescript
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  private apiUrl = 'https://arn:aws:lambda:eu-north-1:211125695713:function:authHandler.execute-api.eu-north-1.amazonaws.com/prod/auth';

  constructor(private http: HttpClient) {}

  login(email: string, password: string): Observable<any> {
    const body = { email, password };

    const headers = new HttpHeaders({
      'Content-Type': 'application/json',
    });

    return this.http.post(this.apiUrl, JSON.stringify(body), { headers });
  }
}
```

Рисунок 3.9 – Компонент для взаємодії з токеном авторизації

У даному розділі було реалізовано систему безпеки для управління завданнями, що включає механізми автентифікації та авторизації користувачів за допомогою протоколу JWT (JSON Web Token), де клієнтська частина на Angular взаємодіє з серверною частиною на AWS Lambda через API Gateway, з використанням Lambda-функції, яка перевіряє облікові дані користувача в базі даних PostgreSQL, генерує токен з інформацією про користувача та його роль, та забезпечує захист доступу до функціоналу відповідно до прав користувачів.

3.4 Тестування компонентів та інтеграційне тестування системи

У цьому розділі буде розглянуто процес тестування та відлагодження вебзастосунку для системи управління завданнями, розробленого з використанням Angular, AWS Lambda та PostgreSQL. Основною метою тестування є забезпечення стабільної, безпечної та передбачуваної роботи системи, а також виявлення та усунення потенційних помилок до впровадження в продуктивне середовище.

При тестуванні системи було залучено наступні види тестування:

– модульне тестування – перевірка окремих функціональних блоків системи, таких як компоненти Angular або окремі методи сервісів

(для цього використовуються Jest (для фронтенду) та unit-тести Node.js (для AWS Lambda));

– інтеграційне тестування – тестування взаємодії між клієнтською частиною (Angular), серверною логікою (AWS Lambda) та базою даних (PostgreSQL) (перевіряється узгодженість та коректність обміну даними);
– системне тестування – охоплює весь застосунок і дозволяє виявити проблеми в логіці, навігації, автентифікації та цілісності даних;
– регресійне тестування – використовується після змін у коді або після усунення помилок для перевірки того, що існуюча функціональність не постраждала;
– навантажувальне тестування – оцінка продуктивності та стабільності системи при різних рівнях навантаження з використанням інструменту Artillery.

Під час тестування було залучено наступні інструменти:

– Jest – фреймворк для модульного тестування Angular-компонентів та сервісів;
– Postman – засіб тестування API-запитів до AWS Lambda через API Gateway;
– Node.js (Jest або Mocha) – для unit- та інтеграційного тестування логіки Lambda-функцій;
– Selenium або Cypress – для автоматизованого тестування UI-функціоналу, такого як авторизація, навігація між сторінками тощо.

Залучено наступні методи відлагодження:

– логування – використання console.log() в Lambda-функціях та інструментів логування Angular для аналізу помилок та взаємодій користувача;
– налагоджувачі – Xdebug для бекенду, а також Chrome DevTools і React Developer Tools для фронтенду;
– статичний аналіз коду – ESLint (JavaScript/TypeScript) та інші лінтери дозволяють виявити помилки ще до запуску застосунку;

– контроль версій – використання Git для управління змінами в коді, створення комітів для окремих етапів тестування, та можливість повернення до стабільної версії при необхідності.

На рисунку 3.10 наведено реалізовану дошку задач проєкту. Ліва частина інтерфейсу містить навігаційне меню, яке дозволяє швидко переходити між основними розділами проєкту: головна, завдання, issue boards, код, тестування, аналітика тощо.

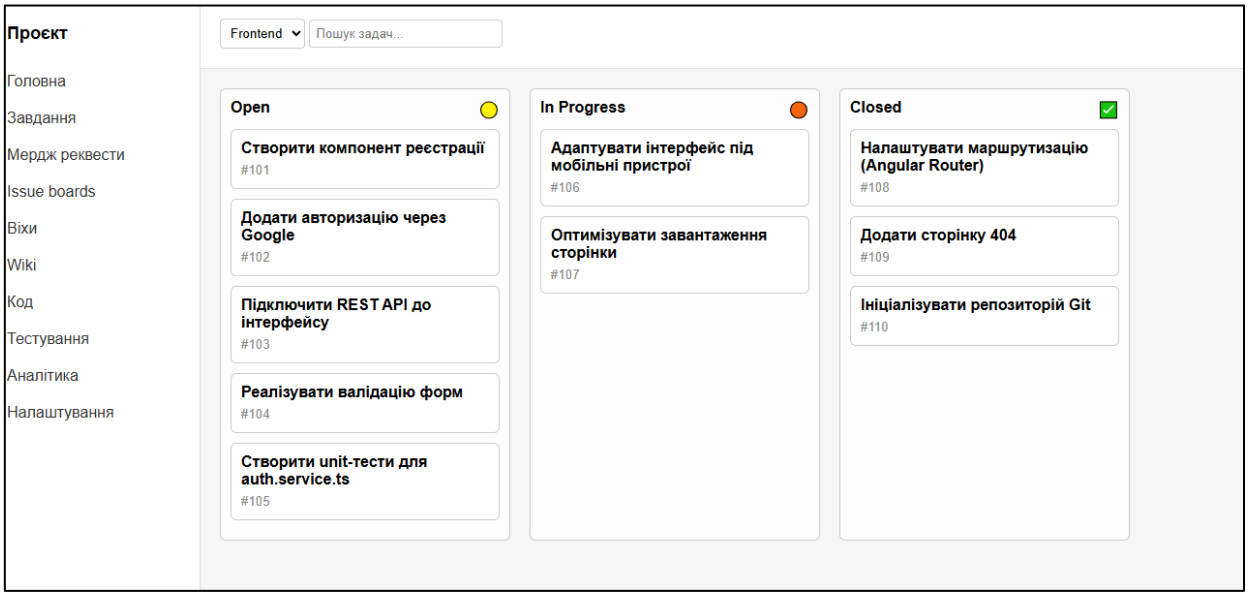


Рисунок 3.10 – Дошка завдань

Центральна частина інтерфейсу поділена на три колонки: Open, In Progress та Closed, які представляють поточний стан задач у проєкті. Кожна задача оформлена у вигляді окремої картки з коротким описом та унікальним номером.

Наприклад, у колонці «Open» є завдання «Створити компонент реєстрації», а в колонці «Closed» – завершене завдання «Налаштувати маршрутизацію (Angular Router)». Такий формат дозволяє ефективно візуалізувати робочий процес команди та контролювати виконання задач на різних етапах розробки.

Якщо аналізувати конкретно про додаткову систему тестування, то варто реалізувати юніт-тести не лише для компонента входу в систему, а й для

інших ключових частин фронтенду, таких як реєстраційна форма, захищені компоненти, а також взаємодія з API.

На рисунку 3.11 наведено компонент реєстрації у застосунку на Angular, створеного з використанням ReactiveFormsModule.

Метою тесту є перевірка правильності логіки компонента RegisterComponent, включаючи перевірку введення, виклик сервісу реєстрації та обробку помилок.

```
TestBed.configureTestingModule({
  imports: [ReactiveFormsModule],
  declarations: [RegisterComponent],
  providers: [
    { provide: AuthService, useValue: mockAuthService },
    { provide: Router, useValue: router }
  ]
}).compileComponents();

fixture = TestBed.createComponent(RegisterComponent);
component = fixture.componentInstance;
fixture.detectChanges();
});

it('should show error if form is invalid', () => {
  component.registerForm.setValue({ email: '', password: '', confirmPassword: '' });
  component.onSubmit();
  expect(component.errorMessage).toBe('');
});

it('should register and navigate to login', () => {
  component.registerForm.setValue({
    email: 'user@test.com',
    password: '123456',
    confirmPassword: '123456'
  });
  mockAuthService.register.mockReturnValue(of({ message: 'Success' }));
  component.onSubmit();
  expect(router.navigate).toHaveBeenCalledWith(['/login']);
});

it('should show error message if registration fails', () => {
  mockAuthService.register.mockReturnValue(throwError({ error: { message: 'User exists' } }));
  component.registerForm.setValue({
    email: 'exist@test.com',
    password: '123456',
    confirmPassword: '123456'
  });
  component.onSubmit();
  expect(component.errorMessage).toBe('User exists');
});
});
```

Рисунок 3.11 – Unit-test реєстрації користувачів

На рисунку 3.12 наведено компонент Unit-test для перевірки роботи функції авторизації, яка реалізована як AWS Lambda-функція.

```
import { handler } from './authHandler.js';
import { SecretsManager } from 'aws-sdk';
import { Client } from 'pg';
import jwt from 'jsonwebtoken';

jest.mock('aws-sdk');
jest.mock('pg');

describe('authHandler integration test', () => {
  it('should return a valid JWT token for correct login', async () => {
    // Моки AWS Secrets
    SecretsManager.prototype.getSecretValue = jest.fn(({ SecretId }) => ({
      promise: async () => ({
        SecretString: JSON.stringify({
          password: 'adminpass',
          JWT_SECRET: 'mysecretkey',
          DB_HOST: 'localhost',
          DB_PORT: 5432,
          DB_NAME: 'mydb',
          DB_USER: 'postgres',
        })
      })
    }));

    // Моки PG-запиту
    Client.prototype.connect = jest.fn();
    Client.prototype.end = jest.fn();
    Client.prototype.query = jest.fn(() => ({
      rows: [{
        user_id: 1,
        login: 'admin',
        password: await bcrypt.hash('123456', 10),
        role_id: 2
      }]
    }));
```

Рисунок 3.12 – Unit-test авторизації

У тесті використовується підхід інтеграційного тестування, де моделюються ключові залежності: AWS Secrets Manager (для отримання конфіденційних даних) і PostgreSQL (для пошуку користувача в базі). Зовнішні сервіси не викликаються безпосередньо, натомість імітуються їхні відповіді.

Логіка перевірки включає:

– успішне отримання конфігурації з секретного сховища AWS;
– підключення до бази даних і знаходження запису користувача за логіном;
– порівняння введеного пароля з тим, що збережений у хешованому вигляді;
– генерація токена доступу у форматі JWT за допомогою секретного ключа;
– перевірка, що згенерований токен дійсний і правильно підписаний.

Як результат, тест гарантує, що якщо дані введені правильно, користувач отримає доступ до системи. Таким чином забезпечується стабільність механізму авторизації перед його запуском у продуктивне середовище.

На рисунку 3.13 наведено сторінку реєстрації користувача. Цей інтерфейс дозволяє новим користувачам створити обліковий запис для доступу в систему завдань.

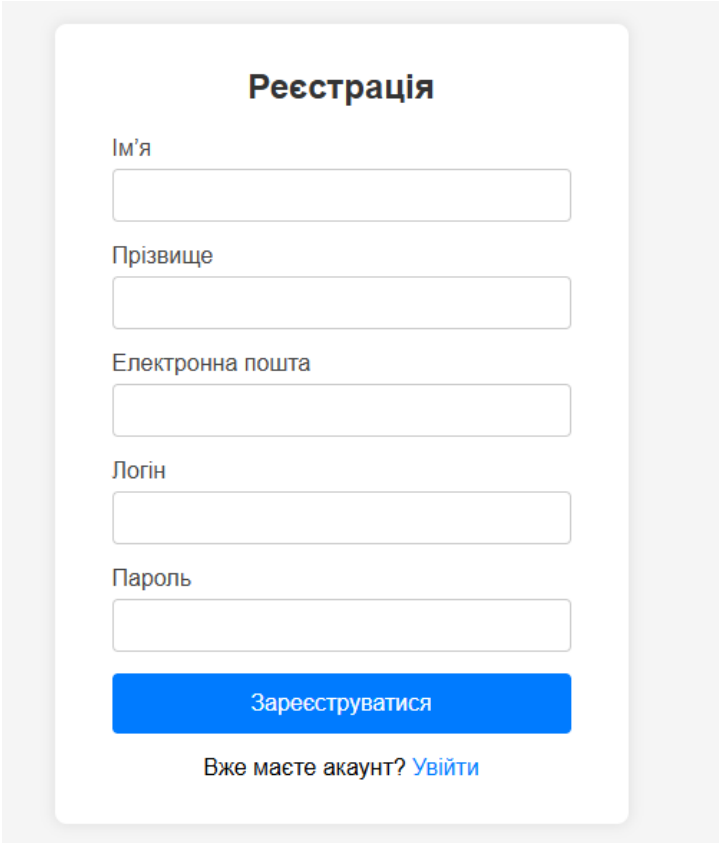


Рисунок 3.13 – Реєстрація користувачів

Для забезпечення високої продуктивності та надійності розроблюваної системи управління завданнями, створеної з використанням Angular на клієнтській частині та AWS Lambda на серверній, проведено тестування навантаження. Основною метою такого тестування було визначення поведінки вебсайту під різними рівнями навантаження, оцінка швидкодії та стійкості системи в умовах одночасної роботи великої кількості користувачів.

Навантажувальне тестування дозволяє оптимізувати ресурси та гарантувати, що сервіс зможе стабільно працювати при реальному

використанні. Для цього застосовувалися сучасні інструменти, такі як Artillery, які підтримують тестування серверних функцій AWS Lambda та фронтенд-додатків на Angular.

На рисунку 3.14 наведено конфігураційний скрипт для навантажувального тестування запитів, що здійснюють отримання даних з бази даних. Даний скрипт налаштований на відправлення GET-запитів до зазначеного API-ендпоінту протягом 60 секунд з інтенсивністю 20 запитів на секунду.

```
config:
  target: 'https://r9j9iupnri.execute-api.eu-north-1.amazonaws.com'
  phases:
    - duration: 60
      arrivalRate: 20
  scenarios:
    - flow:
        - get:
            url: "/u25s6o9"
```

Рисунок 3.14 – Скрипт тестування навантажень

На рисунку 3.15 наведено графік результатів проведеного навантажувального тестування вебсайту. Графік відображає зміну кількості оброблених запитів за секунду (Throughput), а також час відповіді (Response Time) системи протягом усього періоду тестування.

Як видно з графіку, система стабільно обробляла запити з інтенсивністю близько 20 запитів на секунду протягом 60 секунд, при цьому середній час відповіді залишався на рівні, що забезпечує прийнятну швидкодію для користувачів.

Під час тестування не було зафіксовано суттєвих піків затримок або помилок, що свідчить про ефективну роботу сервісу в заданих умовах навантаження. Отримані результати підтверджують здатність системи витримувати заплановані навантаження та забезпечують основу для подальшої оптимізації.



Рисунок 3.15 – Скрипт тестування навантажень

У даному розділі було проведено тестування та відлагодження вебзастосунку для системи управління завданнями, зокрема реалізовано модульне, інтеграційне, системне, регресійне та навантажувальне тестування з використанням Jest, Postman, Node.js, Selenium та Artillery, розроблено методи відлагодження через логування, налагоджувачі та статичний аналіз коду, а також представлено візуальні компоненти системи, включаючи дошку завдань з колонками різних статусів (Open, In Progress, Closed), та продемонстровано реалізацію unit-тестів для компонентів реєстрації та авторизації користувачів.

3.5 Висновки до розділу 3

У третьому розділі було розглянуто процес реалізації та тестування основних компонентів системи управління завданнями, розробленої із застосуванням фреймворку Angular для клієнтської частини та AWS Lambda з використанням Node.js і PostgreSQL – для серверної логіки. Основну увагу приділено інтеграції frontend та backend частин, налагодженню взаємодії з базою даних через AWS Secrets Manager, а також реалізації механізмів авторизації та реєстрації користувачів.

У розділі також охарактеризовано різні типи тестування, включно з модульним, інтеграційним, системним та навантажувальним, які були застосовані для перевірки працездатності кожного компонента системи та оцінки продуктивності в умовах високого навантаження. Було продемонстровано, як за допомогою таких інструментів, як Jest, Jasmine, Node.js, Postman та Artillery, забезпечується висока надійність, безпечність, функціональність та масштабованість системи при обробці до 20 запитів на секунду.

Особливу увагу приділено тестуванню механізму авторизації – ключового елементу безпеки. Завдяки реалізованим юніт-тестам і перевірці взаємодії клієнта з Lambda-функцією авторизації було підтверджено, що система здатна обробляти коректні та некоректні запити, зберігати токени та захищати маршрути.

Таким чином, проведені розробка та тестування доводять, що реалізована система є ефективною, розширюваною та готовою до використання в умовах реального середовища.

ВИСНОВКИ

У кваліфікаційній роботі розроблено систему управління завданнями «TaskManager» засобами фреймворку Angular та з використанням AWS Lambda для серверної частини.

Під час роботи було реалізовано такі задачі:

– проаналізовано аналогічні рішення та технології розробки;
– сформульовано вимоги до системи управління завданнями;
– спроєктовано архітектуру системи та структуру даних;
– реалізовано компоненти клієнтської частини на Angular;
– розроблено серверну логіку з використанням AWS Lambda;
– створено систему автентифікації та авторизації;
– проведено інтеграційне та навантажувальне тестування.

У першому розділі наведено аналіз систем управління завданнями, зокрема Wrike, JIRA, OpenProj. Обґрунтовано вибір Angular для фронтенд-розробки та досліджено можливості AWS Lambda для серверної частини. У другому розділі продемонстровано такі етапи проєктування: концептуальний (діаграми прецедентів, макети інтерфейсу), логічний (діаграми класів та послідовності, схема даних) та фізичний (діаграма компонентів та схема розгортання) рівні. У третьому розділі наведено приклади реалізації компонентів клієнтської частини, серверної логіки на AWS Lambda та системи автентифікації на основі JWT. Наведено результати тестування системи засобами Jest, Postman, Artillery та Selenium.

Результати виконання роботи можуть бути використані для розробки масштабованих та безпечних систем управління завданнями у вебсередовищі та в майбутніх роботах, пов’язаних з безсерверними архітектурами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Топ 10 систем управління проектами для України. URL: https://www.livebusiness.com.ua/ua/tools/pm/ (дата звернення: 11.03.2025).

2. 25 Best Task Management Software Reviewed For 2025. URL: https://thedigitalprojectmanager.com/tools/best-task-management-software/ (дата звернення: 16.03.2025).

3. Day B. Wrike Review: Features, Pros And Cons. *Forbes Advisor*. URL: https://www.forbes.com/advisor/business/software/wrike-review/ (дата звернення: 17.03.2025).

4. Tammy. Jira Pros and Cons in 2025: Features, Pricing & User Reviews. *ONES.com* | Advanced software development management platform for enterprise. URL: https://ones.com/blog/jira-pros-and-cons-in-2025 (дата звернення: 20.03.2025).

5. OpenProj – Project Management. *SourceForge*. URL: https://sourceforge.net/projects/openproj/ (дата звернення: 25.03.2025).

6. What is Angular? URL: https://angular.dev/overview (дата звернення: 28.03.2025).

7. Розуміння зв'язування даних в Angular: Ключ до динамічних веб-додатків – javascript.org.ua – JS Communities. *javascript.org.ua – JS Communities*. URL: https://javascript.org.ua/rozuminnya-zvyazuvannya-danih-v-angular-klyuch-do-dinamichnih-veb-dodatkiv/ (дата звернення: 31.03.2025).

8. Shields K. Angular Development Pros and Cons: Is It Right for Your Web Application. Mobile App & Web Development Greenville, SC | *Designli*. URL: https://designli.co/blog/angular-development-web-application (дата звернення: 05.04.2025).

9. Advantages and Disadvantages of Angular. Tutorials on Technical and Non Technical Subjects. URL: https://www.tutorialspoint.com/angular/angular-advantages-disadvantages.htm (дата звернення: 09.04.2025).

10. 8 Key Benefits of Using Angular for Development. Top Software Outsourcing Company in Vietnam – Orient Software. URL: https://www.orientsoftware.com/blog/benefits-of-angular/ (дата звернення: 12.04.2025).

11. Top 7 Frontend Frameworks to Use in 2025: Pro Advice. URL: https://roadmap.sh/frontend/frameworks (дата звернення: 13.04.2025).

12. Ahmad S. The Pros and Cons of Using React vs. Vue.js vs. *Angular*. *DEV Community*. URL: https://dev.to/seyedahmaddv/the-pros-and-cons-of-using-react-vs-vuejs-vs-angular-1ppk (дата звернення: 16.04.2025).

13. AWS Lambda: Serverless Computing Explained – AWS. URL: https://aws.amazon.com/awstv/watch/64bb50cff05/ (дата звернення: 18.04.2025).

14. What is Amazon VPC? – Amazon Virtual Private Cloud. URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html (дата звернення: 19.04.2025).

15. What is Amazon EC2? – Amazon Elastic Compute Cloud. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html (дата звернення: 19.04.2025).

16. Hosted PostgreSQL – Amazon RDS for PostgreSQL – AWS. *Amazon Web Services, Inc*. URL: https://aws.amazon.com/rds/postgresql/ (дата звернення: 20.04.2025).

17. Welcome to AWS Documentation. URL: https://docs.aws.amazon.com/ (дата звернення: 20.04.2025).