МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СИСТЕМИ АВТОРИЗАЦІЇ НА ОСНОВІ РОЗПІЗНАВАННЯ ОБЛИЧЧЯ»**

Виконав: студент 2 курсу, групи 8.1212-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
В.В. Тимощук
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, PhD Столярова А.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Тимощуку Віктору Володимировичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи авторизації на основі розпізнавання обличчя

керівник роботи Столярова Анастасія Валеріївна, PhD
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » травня 2023 року № 642-с

2. Строк подання студентом роботи 27.11.2023 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка системи авторизації на основі розпізнавання обличчя.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 03.05.2023 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.	15.05.2023	
2.	Збір вихідних даних.	01.06.2023	
3.	Обробка методичних та теоретичних	30.06.2023	
	джерел.		
4.	Розробка першого та другого розділу.	28.08.2023	
5.	Розробка третього розділу.	06.11.2023	
6.	Оформлення та нормоконтроль	20.11.2023	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	15.12.2023	

Студент _____ В.В. Тимощук
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.В. Столярова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка системи авторизації на основі розпізнавання обличчя»: 88 с., 30 рис., 11 джерел, 3 додатка.

AI, ANGULAR, AWS, COMPUTER VISION, HTTP, MONGODB, MONGOOSE, NESTJS, RECOGNITION, RXJS, S3, TAILWIND CSS, TYPESCRIPT.

Об’єкт дослідження – процес розробки системи авторизації.

Мета роботи: розробити систему авторизації на основі розпізнавання обличчя з використанням технологій фреймворків Angular, NestJS та сервісів Amazon.

Методи дослідження – методи аналізу програмного забезпечення та сервісів, методи класифікації об’єкту розробки, методи моделювання, проєктування, конструювання та тестування програмного застосунку.

В процесі розробки програмного застосунку було проаналізовано обрані для розробки технології, досліджено та опрацьовано особливості предметної області, за допомогою діаграм та блок-схем було спроєктовано архітектуру програмного застосунку, його функціонал, поведінку тощо. Для розробки клієнтської частини застосунку було використано фреймворк Angular, для серверної частини – фреймворк для бекенд розробки NestJS. Обробку та аналіз зображень виконував натренований інструмент штучного інтелекту Amazon Rekognition.

Результатом розробки виступає сучасна, проста у використанні та доволі захищена система авторизації, яка дозволить клієнтам тієї чи іншої системи проводити миттєву авторизацію в застосунку, використовуючи лише біометричні дані користувача.

SUMMARY

Master’s qualifying paper «Development of the Authorization System Based on the Face Recognition»: 88 pages, 30 figures, 11 references, 3 supplements.

AI, ANGULAR, AWS, COMPUTER VISION, HTTP, MONGODB, MONGOOSE, NESTJS, RECOGNITION, RXJS, S3, TAILWIND CSS, TYPESCRIPT.

The object of the study is the process of developing the authorization system.

The aim of the study is to develop the authorization system based on face recognition using Angular, NestJS framework technologies and Amazon services.

The methods of research are software and service analysis methods, development object classification methods, software application modeling, design, construction and testing methods.

In the process of developing the software application, we analyzed the technologies selected for development, researched and worked out the features of the subject area, and designed the architecture of the software application, its functionality, behavior, etc. using diagrams and flowcharts. The Angular framework was used to develop the client side of the application, and the NestJS backend framework was used for the server-side development. Image processing and analysis was performed by a trained Amazon Rekognition artificial intelligence tool.

The result of the development is a modern, easy-to-use and fairly secure authorization system that will allow customers of one or another system to perform instant authorization into the application using only the user’s biometric data.

ЗМІСТ

ВСТУП

Сучасний світ передбачає стрімкий розвиток технологій в абсолютно будь-якій сфері нашого життя, чи то медицина, виробництво, наука або освіта, чи то банальний побут, та всі його складові. Технології щільно оточують нас і йдуть з нами крок за кроком, допомагаючи нам та спрощуючи наше життя в цілому.

Якщо ми говоримо про технології, які людство використовує в повсякденному житті, то перше, що нам приходить на думку – це телефон, комп'ютер, ноутбук чи планшет – речі без яких сьогодення здається просто неможливим. І з цим складно не погодитись, наприклад магазин, спортзал або навіть лікарня, розваги, та різного роду сервіси буквально завжди знаходяться у нас під рукою. Безперечно, бізнес активно використовує розвиток технологій для свого просування на ринку послуг та інформації, таким чином ефективно привертаючи увагу ще більшої кількості клієнтів та покращуючи їхній досвід взаємодії з сервісами, продуктами або послугами, які надає цей бізнес.

Однак, мабуть, кожен із вас зіштовхувався з тим, що при використанні того чи іншого сервісу, вас в необхідному порядку зобов'язують поділитися персональними даними, це може бути ваше ім'я та прізвище, номер телефону, електронна пошта та інше. Таким чином, бізнес ідентифікує вас задля подальшої можливості контактувати з вами, надавати послуги, робити сповіщення, тощо. Повертаючись до стрімко зростаючих технологій, ми маємо і зростаючі ризики їх використання, тому що в будь-якому бізнес-застосунку чи сервісі який ви використовуєте, ваші персональні дані можуть бути викраденими і використаними в намірах злочинного характеру. І тому бізнес намагається як можна краще захистити дані користувача і не наражати його на небезпеку, використовуючи всі можливі засоби захисту даних. Одними з таких методів є захист користувацького акаунту паролем, верифікація входу у власний акаунт через сповіщення на телефон або листом на вашу електронну пошту, і доволі сучасний метод – використання біометричних даних для авторизації користувача

в систему. Останній може мати на увазі вхід в систему за допомогою відбитку пальців або використанням біометричних даних вашого обличчя. Цей спосіб авторизації в сучасності дуже поширений, його перевагою є використання унікальних біометричних даних людини, які майже неможливо сфальсифікувати. Однак, у такого способу авторизації є дуже великий недолік – його імплементація обумовлена великими витратами розробницьких та фінансових ресурсів. Тому, його використовують тільки великі компанії, які можуть собі це дозволити, наприклад: Microsoft, Google, Apple, Xiaomi, та інші популярні виробники різної категорії пристроїв. Очевидно, малий та середній бізнес не може собі цього дозволити і тому це становить собою серйозну проблему. Покращення захисту даних клієнтів, спрощення процесу та скорочення витраченого часу на вхід в систему гарантовано відгукнеться в поліпшенні лояльності та довіри до бізнесу, що являється головною запорукою успішної діяльності будь-якого закладу чи підприємства.

Актуальність роботи визначається доволі обмеженим використанням систем авторизації на основі розпізнавання обличчя у колі розробників та у бізнесі. Вебсайти та онлайн сервіси, які ми використовуємо кожен день, не використовують даний спосіб авторизації, але впровадження такого могло б покращити досвід використання та швидкість входу в систему.

Наукова новизна визначається тим, що система яка буде розроблена – безперечно вирішить проблему обмеженого поширення авторизації. Вона матиме покращену конфігурацію для розпізнавання облич, що дозволить проводити процес ідентифікації точно та без помилок.

Об’єкт дослідження – процес розробки системи авторизації.

Предметом дослідження є процеси ідентифікації досліджуваних методів авторизації, їх характеристика, а також проблеми та способи їх вирішення.

Отже, **мета** цієї роботи – створення системи авторизації на основі розпізнавання обличчя та вирішення проблеми доступності цього способу авторизації та сприяння його поширенню. Очікуваний результат роботи цієї системи полягає в поліпшенні захисту конфіденційних даних користувачів того

чи іншого бізнесу, прискорення та спрощення процесу авторизації в систему без втрат показника надійності та безпеки.

Для досягнення поставленої мети необхідно вирішити наступні **завдання**:

– провести аналіз більшості сучасних методів авторизації;
– дослідити актуальність предметної області та її проблеми;
– обрати технології для розробки;
– розробити діаграму класів та послідовностей для деталізації архітектури системи та ефективної розробки;
– створити дизайн системи;
– реалізувати систему авторизації на основі досліджень та проєктування.

Методи дослідження – методи аналізу програмного забезпечення та сервісів, методи класифікації об’єкту розробки, методи моделювання, проєктування, конструювання та тестування програмного застосунку.

Практична значущість роботи полягає у можливості використання результату розробки (систему авторизації) у більшості сучасних онлайн-сервісів та вебдодатках, які кожен день використовують люди в різних сферах нашого життя. Таким чином, система авторизації може вплинути на наступні аспекти: покращення лояльності користувачів до того чи іншого сервісу, покращення швидкості ідентифікації користувачів та збільшення рівня захищеності тої чи іншої системи

Структура роботи: робота складається з вступу, трьох розділів, висновків, списку використаної літератури та трьох додатків.

У вступі подано загальні відомості про дану наукову роботу, починаючи від умотивування теми, мети, завдань, актуальності дослідження, визначення об’єкту, предмету та структурування роботи.

У першому розділі було проведено опис та характеристика систем авторизації, досліджено та проаналізовано більшість сучасних систем авторизації, визначено проблеми таких систем, описано особливості предметної області та розглянуто обрані технології для розробки системи авторизації.

Другий розділ присвячений проєктуванню системи. У цьому розділі було

розглянуто побудовані діаграми та їх значимість у процесі розробки програмних застосунків, оглянули створений дизайн системи та його важливість на етапі проєктування.

Третій розділ містить в собі детальний опис інструментів розробки, розгляд результатів розробки клієнтської та серверної частини додатку, а також результати тестування.

У висновках подано узагальнені результати проведеної роботи.

Загальна кількість сторінок 88, кількість використаних джерел 11.

1 ОГЛЯД ТЕХНОЛОГІЙ, ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ПРОБЛЕМ

1.1 Поняття та характеристика систем авторизації

Системи авторизації – це комплексні механізми, спрямовані на підтвердження ідентичності користувачів та надання доступу до певних ресурсів або функціоналу системи. Основні компоненти систем авторизації включають в себе процес аутентифікації, контроль доступу та управління привілеями.

Її ефективність визначається рівнем безпеки, гнучкістю, тобто можливістю пристосуватися до змін, і зручністю для користувачів.

Гнучкість системи авторизації повинна передбачати її адаптивність до змін вимог, що виникають під час технологічного розвитку або від потреб користувачів. Здатність змінюватись, розширюватись та інтегруватись з іншими системами стає ключовим аспектом.

Зручність для користувачів означає простоту та доступність у використанні системи. Інтерфейс авторизації повинен бути зрозумілим та швидким, щоб спростити процес для кінцевого користувача.

Ці аспекти дозволяють системі впевнено працювати у сучасному цифровому середовищі, гарантуючи не лише безпеку, а й зручність та гнучкість використання.

1.2 Огляд існуючих систем авторизації

Забезпечення безпеки та захисту персональних даних у сучасному цифровому середовищі вимагає розгляду та використання різноманітних систем авторизації. У цьому розділі ми розглянемо основні типи систем, які використовуються для підтвердження ідентичності користувачів та подальшого

надання доступу до ресурсів. Кожен з цих типів має свої унікальні особливості, переваги та обмеження, які варто враховувати при виборі оптимального методу авторизації для конкретної системи чи проєкту.

1.2.1 Парольні системи авторизації

Парольна система авторизації є однією з найпоширеніших та найбільш зрозумілих для користувачів. Вона базується на введенні ключової комбінації символів (пароля) для перевірки ідентифікації користувача та надання доступу до системи або ресурсів. Перші паролі з'явилися у середовищі комп'ютерів у 1960-х роках, коли виникла потреба у способі захисту від несанкціонованого доступу.

Переглянемо основні характеристики парольних систем.

Простота використання. Паролі є простими у запам’ятовуванні та введенні, що робить їх зручними для багатьох користувачів.

Вразливість до атак. Однак, паролі можуть бути піддані атакам зламу, особливо якщо вони слабкі або їх легко вгадати. Недостатня складність паролів може призвести до їхнього підбору або зламу.

Необхідність регулярно змінювати паролі. Щоб підтримувати безпеку, рекомендується регулярно змінювати паролі. Але це може призвести до проблем із запам'ятовуванням та накопиченням слабких паролів.

Технологічні вдосконалення. Для підвищення безпеки використання паролів застосовуються методи двофакторної аутентифікації, які вимагають введення коду з SMS або використання апаратних пристроїв для підтвердження ідентифікації.

Керування паролями. Управління та збереження безпечних паролів стає важливим аспектом. Використання менеджерів паролів або вимога до складних комбінацій може покращити безпеку.

Як бачимо, використання парольних систем має більше недоліків аніж

позитивних рис, таких як зрозумілість у використанні та легкість у впроваджені даного типу технології в бізнес-систему. Сьогодні вищезазначені недоліки легко перекриваються наступними способами авторизації.

1.2.2 Мультифакторна аутентифікація

Мультифакторна аутентифікація (MFA) – це процес перевірки ідентифікації користувача за допомогою двох або більше методів, що забезпечує вищий рівень безпеки, порівняно з традиційними однофакторними методами, наприклад, паролем.

Перейдемо до основних характеристики мультифакторної аутентифікації.

Додатковий рівень безпеки. Комбінація двох або більше методів аутентифікації створює вищий рівень безпеки, оскільки навіть якщо один метод компрометовано, інші залишаються недоступними для несанкціонованого доступу.

Різноманітні методи. MFA може включати в себе безліч варіантів послідовної ідентифікації користувача, це можуть бути: паролі, фізичні токени або смарт-карти, одноразові електронні коди/токени тимчасової валідності, згенеровані зовнішніми сервісами, біометричні дані, такі як відбиток пальця чи розпізнавання обличчя та інші.

Зручність і безпека. MFA, хоча і надає вищий рівень безпеки, але може бути реалізована з урахуванням зручності для користувачів. Наприклад, застосування відбитку пальця для розблокування ноутбуку чи телефону, або ж мобільних додатків, які генерують одноразові коди для входу в систему того чи іншого бізнесу.

Широке застосування. Завдяки своїй ефективності, MFA використовується в багатьох сферах: в банківській справі, хмарних сервісах, доступі до корпоративних мереж тощо.

MFA вважається ефективним засобом підвищення безпеки і надійності в

сучасних умовах зростаючих кіберзагроз. Його впровадження може значно зменшити ризики несанкціонованого доступу та компрометації особистої інформації.

1.2.3 Біометричні системи авторизації

Біометричні системи авторизації використовують фізіологічні або поведінкові характеристики для ідентифікації та аутентифікації користувачів. Ці системи ґрунтуються на унікальних особистих рисах людини, які є унікальними та відмінними для кожної особи.

Розглянемо загальні характеристики біометричної системи авторизації.

Фізіологічні характеристики. Вони базуються на фізичних атрибутах, таких як відбитки пальців, розпізнавання обличчя, риси руки (пальців), біометрія ока тощо.

Поведінкові характеристики. Ці системи використовують такі унікальні риси, як почерк, голосові параметри або стиль набору тексту.

Високий рівень безпеки. Біометричні дані важко або практично неможливо підробити або ж втратити, що робить цю систему дуже безпечною для авторизації.

Зручність використання. Крім високого рівня безпеки, вони забезпечують також зручність використання, оскільки вимагають лише присутності користувача для ідентифікації.

Активне та пасивне використання. Деякі системи можуть використовувати біометричні дані активно (наприклад, введення відбитків пальців) або пасивно (наприклад, розпізнавання обличчя без активної дії користувача).

Мультифакторна аутентифікація. Біометричні дані часто використовують як один із факторів у системах мультифакторної аутентифікації для підвищення рівня безпеки.

Біометричні системи авторизації є ефективним та досить надійним методом ідентифікації особи, проте їх успішність може залежати від точності та якості зчитування біометричних даних, а також від збереження та захисту цих даних від несанкціонованого доступу.

1.2.4 Блокчейн авторизація

Авторизація на основі розподіленої бази даних, яка базується на технології блокчейн, використовує принципи децентралізації та криптографії для контролю доступу до інформації чи ресурсів. Блокчейн – це система, в якій дані зберігаються у блоках, кожен з яких зв'язаний з попереднім за допомогою криптографічних хеш-функцій, що створює ланцюг блоків. Авторизація на основі блокчейну здійснюється шляхом управління правами доступу через цей розподілений та недоступний для змін блокчейн.

Переглянемо основні характеристики блокчейн авторизації.

Децентралізація. Інформація зберігається на кількох вузлах мережі, що робить блокчейн відкритим і відомим усім користувачам. Кожен блок має підпис, що відображає його унікальність і запобігає внесенню змін.

Криптографія та безпека. Блокчейн використовує криптографічні методи для захисту даних, що забезпечує високий рівень безпеки. Кожен користувач отримує свій унікальний ключ, який контролює його доступ до інформації.

Смарт-контракти. Це програми, що автоматизують виконання угод та інших операцій на блокчейні. Вони можуть використовуватися для автоматичної авторизації користувачів за певних умов.

Недоступність змін. Один раз внесені дані вже не можуть бути змінені або вилучені без згоди декількох учасників мережі, що робить цю систему надійною та відстежуваною.

Прозорість та аудит. Блокчейн забезпечує можливість перевірки всіх здійснюваних операцій, що робить його ідеальним для аудиту та створення систем з високим рівнем довіри.

Блокчейн авторизація надає високий рівень безпеки, надійності та прозорості в управлінні доступом до інформації або ресурсів. Вона знаходить широке застосування у фінансових послугах, логістиці, медицині та інших галузях, де важлива конфіденційність та безпека даних.

В свою чергу має доволі неінтуїтивний процес впровадження в бізнес-систему та потребує підвищеної кваліфікації IT-спеціаліста який буде займатись імплементацією даної технології.

1.2.5 OAuth та OpenID авторизація

OAuth (Open Authorization) – це протокол авторизації, призначений для надання доступу третім додаткам до ресурсів, які контролюються іншими службами (постачальниками). Основна мета полягає в тому, щоб дозволити користувачам давати обмежений доступ до своїх ресурсів, таких як дані профілю в соціальних мережах або електронної пошти, без необхідності передавати свій пароль третім сторонам.

Звернемо увагу на основні характеристики OAuth авторизації.

Авторизація без передачі пароля. OAuth дозволяє користувачам надавати доступ до своїх даних чи ресурсів без передачі пароля стороннім додаткам. Замість цього, використовуються токени доступу, які видаються користувачеві на відведений проміжок часу.

Обмежений доступ до користувацьких даних. Користувач має можливість обирати, які конкретні ресурси чи дані будуть доступні третім додаткам під час авторизації через OAuth.

Механізм токенів доступу. OAuth використовує токени доступу для підтвердження авторизації. Ці токени можуть мати обмежений строк дії та обмежений доступ до конкретних ресурсів.

Різноманітні версії та типи. Існує декілька версій протоколу OAuth, таких як OAuth 1.0, OAuth 2.0, які мають свої унікальні особливості та переваги. OAuth 2.0, наприклад, є більш універсальним та дозволяє робити авторизацію з

використанням різних механізмів (соціальні мережі, електронна пошта, сторонні провайдери та навіть блокчейн авторизація).

OAuth авторизація стала на сьогодні стандартом для безпечного та зручного обміну даними між різними додатками та сервісами. Цей тип авторизації дозволяє користувачам контролювати доступ до своїх ресурсів, забезпечуючи при цьому високий рівень безпеки та зручності використання.

Наступним буде метод, який часто плутають з OAuth, це OpenID.

OpenID – це протокол одноразової ідентифікації, який дозволяє користувачам використовувати один цифровий ідентифікатор для авторизації на різних вебсайтах чи сервісах без необхідності створення окремих облікових записів для кожного з них.

Переглянемо положення характерні OpenID авторизації.

Цифровий ідентифікатор. OpenID використовується для створення цифрового ідентифікатора, який є унікальним для кожного користувача. Цей ідентифікатор може бути використаний для авторизації на різних сайтах, які підтримують протокол OpenID.

Універсальний логін. Користувач може використовувати один цифровий ідентифікатор (OpenID) для входу на різні вебсайти, які підтримують цей протокол. Це уникне необхідності створення окремого облікового запису на кожному сайті.

Спрощена реєстрація. За допомогою OpenID користувач може швидко увійти на новий вебсайт, не реєструючи новий обліковий запис, що зменшує кількість паролів та ідентифікаторів, які потрібно пам'ятати.

OpenID забезпечує більш зручний та безпечний спосіб авторизації на вебсайтах, спрощуючи процес входу для користувачів та відмінно впливаючи на безпеку та захист їхніх даних.

Головна різниця між вищезазначеними технологіями полягає в тому, що OpenID використовується для ідентифікації користувача, тоді як OAuth – для надання доступу до ресурсів чи даних. Іноді ці два протоколи можуть використовуватись разом: OAuth використовується для отримання доступу до ресурсів, а OpenID для ідентифікації користувача.

1.3 Проблеми систем авторизації сьогодні

Ознайомившись з основними існуючими системами авторизації, стає зрозумілим що вибір технологій для ідентифікації користувача і водночас захисту його персональних даних сьогодні є дуже і дуже широким.

Однак, системи авторизації стикаються з рядом викликів, що ускладнюють їх безпеку та ефективність. Однією з основних проблем є рівень уразливості. Вразливості аутентифікації, такі як фішинг або перехоплення токенів, викликають загрозу для безпеки даних. Недоліки у системах авторизації можуть використовуватись зловмисниками для несанкціонованого доступу та порушення конфіденційності.

Також, важливим аспектом є баланс між безпекою та зручністю. Часто заходи безпеки можуть впливати на зручність використання. Нерідко ускладнення процесу авторизації може призвести до того, що користувач просто втомиться від вводів паролю, особливих слів, запитань, очікувань кодів на пошту та телефон, тому зручність і простота становить дуже високу значимість у системах авторизації.

Крім того, управління конфіденційною інформацією також є складною задачею. Зберігання та контроль інформації про ідентифікацію користувачів вимагає високого рівня уваги до деталей, оскільки будь-яке порушення цих даних може призвести до серйозних проблем безпеки та приватності.

1.4 Опис предметної області та системи

В даній роботі предметною областю виступає розробка системи авторизації на основі розпізнавання обличчя. Вона буде реалізована у вигляді вебдодатку, який користувач зможе відвідати методом вводу посилання на сторінку авторизації в браузері. Даний вебдодаток буде складатись зі сторінки авторизації користувача методом вводу зображення обличчя, а також сторінки,

яка містить адмін-панель.

Панель адміністратора включає в себе сторінку з таблицею відображення зареєстрованих користувачів і сторінку, де адміністратор може внести нового користувача до системи.

Сторінка з таблицею відображення зареєстрованих користувачів являє собою список користувачів, коротку інформацію про того чи іншого користувача, можливості його видалити, а також оновити вміст таблиці.

Сторінка реєстрації нового користувача має вигляд форми з деякими полями вводу, а також блоком для завантаження фотографії або фотографій користувача в анфас.

Процес авторизації користувача проходить в наступному порядку: користувач заходить на сторінку авторизації, дозволяє браузеру використовувати камеру поточного пристрою, система бере знімок обличчя і проводить необхідні обчислення, в результаті яких відображається статус ідентифікації користувача, після чого дозволяє або забороняє вхід в систему.

1.4.1 Опис системи

Авторизація у вебдодаток чи будь-який інший онлайн сервіс за допомогою розпізнавання обличчя являється дуже важливим і необхідним у використанні способом на сьогодні. Авторизація за допомогою розпізнавання обличчя людини використовується з міркувань безпеки та зручності використання для кінцевого користувача.

Розпізнавання обличчя вважається високобезпечним методом авторизації через унікальність кожного обличчя. Наприклад, системи розпізнавання обличчя вже успішно використовуються у банківській сфері, де вони забезпечують доступ до особистих фінансових даних, наприклад, як у «Apple Face ID» чи «Samsung Face Recognition».

Використання розпізнавання обличчя забезпечує швидкий та зручний

доступ без необхідності запам’ятовувати паролі. Наприклад, мобільні пристрої використовують розпізнавання обличчя для розблокування екрану, як у пристроях на базі Android чи iOS.

Окрім того, перевага такого способу авторизації полягає у тому, що користувачеві не потрібно нічого робити – достатньо просто показати обличчя. Це особливо зручно в сферах, де важливо максимально зменшити час на аутентифікацію, наприклад, на вході до приміщень через системи розпізнавання обличчя.

З огляду на вищевказані твердження і дивлячись на те, як сьогодні стрімко розвиваються нейронні мережі, в процесі цієї роботи ми розробимо сучасну та робочу систему авторизації на базі розпізнавання обличчя, яке буде виконуватись нейронною мережею.

Звичайно, як вже було зазначено, дані системи авторизації вже працюють і використовуються в бізнесі. Наприклад, цю систему широко використовують компанії Apple, Microsoft, Samsung, LG, Huawei та інші. Проблема, яка постає перед нами, полягає в дуже малій поширеності використання такого способу малими та середніми бізнесами. У мережі Інтернет практично неможливо знайти поширений пакет для розпізнавання обличчя, який був би імплементований або перебував на стадії продакшн.

Взявши до уваги вищезазначене, постає завдання в розробці такої системи, за допомогою якої можна буде впроваджувати авторизацію на основі розпізнавання обличчя в абсолютно будь-якому вебдодатку.

1.5 Функціональні та нефункціональні вимоги до системи

Перед тим як почати імплементацію системи авторизації на основі розпізнавання обличчя слід утвердити модель поведінки додатку та нефункціональні вимоги.

Для формування основи проєкту нам потрібно сформулювати та

дотримуватись впродовж розробки наступних функціональних вимог:

– менеджмент облікових записів: система повинна давати можливість адміністратору створювати, переглядати та видаляти облікові записи;
– авторизація: система авторизації повинна дозволяти користувачу проходити етап ідентифікації, використовуючи знімок обличчя зчитаний з камери пристрою;
– акаунт користувача: система повинна давати можливість переглядати вміст користувацього акаунту, а також надавати змогу завершити сесію та пройти процес авторизації знову.

Визначившись з функціональними вимогами системи авторизації, ми чітко розуміємо, яка задача постає перед розробником, дозволяючи уникнути непередбачених ситуацій і продумування функціоналу системи «на льоту». Це дозволить максимізувати продуктивність розробки, а також сконцентрувати увагу на важливих деталях.

До нефункціональних вимог системи ми віднесемо наступні пункти:

– безпека: система повинна працювати в підвищеному рівні безпеки, так як під час авторизації опрацьовуються та зберігаються біометричні дані людини, підробка або втрата яких може привести до серйозних наслідків розгорнута система повинна зберігати секретні ключі, та інші параметри конфігурації в надійному та недоступному для сторонніх суб’єктів місці);
– швидкодія: система повинна виконувати та опрацьовувати всі запити до серверу не більше 2 секунд;
– інформативність: система повинна безперечно точно інформувати користувача про статус авторизації та всі зміни стану системи, динамічно змінюючи вигляд сторінки проявленням текстової інформації, спливаючих вікон та відображення іконок-спінерів, які означають завантаження або обробку даних (панель адміністратора повинна мати вичерпний опис полів вводу даних та файлів, для коректної реєстрації облікового запису, а також достатньо інформації

для ідентифікації того чи іншого користувача на сторінці таблиці користувачів);

– кросплатформеність та кросбраузерність: система повинна бути доступною на будь-яких пристроях, чи то на персональному комп’ютері, чи то на смартфоні (це стосується і веббраузерів, додаток повинен коректно відображатись на всіх сучасних браузерах).

1.6 Огляд технологій та інструментів

1.6.1 Angular

Angular – це прогресивний фреймворк для розробки вебдодатків, який пропонує високорівневі концепції та інструменти для створення сучасних рішень [1]. Його основна концепція базується на компонентній архітектурі, де різні елементи вебдодатку, такі як кнопки, форми, чи списки, представлені у вигляді відокремлених компонентів. Це спрощує структуру програми, роблячи код більш організованим та простим у розумінні.

Для прив’язки даних Angular використовує механізм Two-Way Binding, котрий спрощує роботу з формами та полями вводу даних. Оновлюючи дані через користувацький інтерфейс, атрибут компоненту (функціональна частина модулю) прив’язаний до поля вводу відразу оновиться вслід за вводом користувача, і навпаки.

Також, один з ключових моментів – це вбудований механізм Dependency Injection. Цей механізм дозволяє керувати залежностями проєкту та створювати взаємодіючі сервіси, що полегшує тестування та розвиток додатків.

Angular надає дуже багато можливостей прямо із коробки, наприклад: динамічний роутинг для реалізації багатосторінкових вебдодатків і комплексної маршрутизації, HTTP Client який дозволяє робити HTTP-запити відразу після створення проєкту, без потреби у встановленні сторонніх бібліотек по типу Fetch

або Axios, валідація форм, потужний інструмент впровадження анімацій та багато інших інструментів для розробки швидких і сучасних рішень.

Окрім того, Angular постає перед нами як кросплатформений фреймворк. Angular в комбінації з технологіями Ionic або NativeScript надає можливість конвертувати вебдодаток написаний мовою TypeScript [2] в повноцінний Android чи iOS застосунок.

1.6.2 NestJS

NestJS – фреймворк для бекенд розробки зі стрімко зростаючою популярністю [3]. Розроблений на платформі NodeJS, написаний та повністю підтримує мову програмування TypeScript. Був обраний для розробки, так як використовує концепції фреймворку Angular. Він переймає концепцію використання модульної архітектури, а також унаслідував механізм Dependency Injection.

Із особливостей, фреймворк має вбудовану підтримку HTTP/WS Middleware, що дозволяє ефективно обробляти HTTP запити та передачу даних через WebSocket з’єднання. Наявність CLI відіграє велику роль в автоматизації створення шаблонів, модулів та інших структурних елементів додатків

NestJS має гнучку та розширювану підтримку різних баз даних. Він може працювати як з SQL базами, так і з NoSQL базами даних.

Для взаємодії з реляційними базами даних, NestJS використовує TypeORM, що спрощує роботу з базами даних через мову програмування. Це дозволяє створювати моделі даних, використовуючи класи та декоратори, забезпечуючи зручний доступ до даних та їх обробку.

У випадку NoSQL баз даних, NestJS має модулі, які спрощують взаємодію з базами, такими як MongoDB, котра і буде використана під час розробки системи авторизації. Влаштовані модулі дозволяють легко підключатися до бази даних та використовувати її прямо в коді бекенд додатку, використовуючи

сучасні підходи та методи роботи з NoSQL.

Для комфортної взаємодії будь-якого розробника з бекенд-додатком розробленим на NestJS, у фреймворку є вбудована підтримка автоматичного генерування API документації на основі робочого коду, котрий помічається спеціальними декораторами, які можна модифікувати, додавши опис API-маршруту та типізацію.

1.6.3 Amazon Rekognition та Amazon S3

Amazon Rekognition – це сервіс, який надає компанія Amazon Web Services (AWS), він спрямований на аналіз та обробку зображень і відео, наприклад, для розпізнавання облич, тексту, якихось об’єктів, також за допомогою нього можна виявити емоції, аксесуари на голові людини, та особливості обличчя людини, що бездоганно підходить під поставлену задачу [4]. Amazon Rekognition має дуже широку та гарно описану документацію, а також дуже комфортні для створення запитів пакети, при встановленні яких, у розробника з’являється повноцінний набір засобів та утиліт, який допомагає використовувати сервіс максимально ефективно, а сама розробка стає зрозумілою та легко розширюваною.

Даний сервіс може приймати вхідні дані, такі як відео або зображення аж трьома способами:

– передаючи зображення у форматі base64;
– передаючи назву зображення або відео файлу який попередньо був завантажений на хмарне сховище S3;
– передаючи потокове відео, обробляючи відеоматеріал в режимі реального часу.

В одному із пунктів було згадане хмарне сховище S3, мова йде про масштабоване сховище даних Amazon S3 (Simple Storage Service), своєрідна база даних компанії Amazon, яка використовується абсолютно всіма сервісами цієї компанії, наприклад зберігання зображень чи відео, файлів того чи іншого

проєкту, або навіть ваша поштова скринька, де будуть зберігатись ваші листи та інша інформація. Amazon Rekognition надає дуже зручний конструктор запитів, де можна з легкістю вказати назву файлу, назву сховища і тип операції, який потрібно виконати з вказаним файлом. Таким чином і буде відбуватись аналіз зображення та розпізнавання обличчя користувача.

2 ПРОЄКТУВАННЯ СИСТЕМИ АВТОРИЗАЦІЇ

Одним із найважливіших кроків під час розробки будь-якого проєкту являється проєктування системи та написання вичерпної документації. Цей етап допоможе встановити чіткі вимоги до проєкту та систематизувати розробку. Для проєктування архітектури, моделі поведінки та модульних залежностей системи буде використано сервіс Draw.io, який має широкий функціонал та велику кількість зручних для використання пресетів.

2.1 Діаграма класів

Діаграма класів – це вид діаграм, який дозволяє візуалізувати структуру класів у системі так, як би вона вигляділа у програмному коді. Вона демонструє зв’язки між класами, їхні атрибути та методи. Атрибути характеризують об’єкти класу, а методи відображають функціонал класу, який взаємодіє з атрибутами класу або ж вхідними даними.

Взаємозв’язки між класами являються однією із найважливіших складових діаграми класів. Вони показують різні типи взаємодії: асоціації, спадкування, агрегації та композиції. Наприклад, асоціація вказує на зв’язок між двома класами, спадкування показує утворення підкласу від іншого класу, агрегація описує взаємозв’язок «частина-ціле», а композиція вказує на залежність «частина-ціле» з обмеженим життєвим циклом.

Використання діаграми класів впливає на покращене розуміння архітектури майбутньої програми, спрощує аналіз зв’язків між класами та допомагає виявити потенційні проблеми на ранньому етапі проєктування програмного забезпечення.

2.1.1 Опис класів системи авторизації

Давайте переглянемо класи системи авторизації та їх роль у майбутньому застосунку. Слід зазначити, що описані далі класи будуть використані безпосередньо під час розробки, так як вони диктують майбутню архітектуру системи, її компоненти, а також атрибути цих компонентів і їх взаємодію між собою всередині системи.

Отже, перейдемо до ознайомлення з класами системи авторизації.

Клас: Адміністратор. Даний клас представляє інформацію про користувача системи з роллю адміністратора. Він має атрибути, такі як ім’я та ID, яких достатньо для ідентифікації адміністратора в силу того, що панель адміністратора в якій він буде працювати буде приватною частиною додатка. Цей клас містить методи, які дозволять вводити необхідні дані для створення облікового запису, створювати облікові записи, переглядати їх та видаляти. Даний клас виконує функцію адміністратора системи та відіграє важливу роль у авторизації майбутніх клієнтів системи.

Клас: Панель адміністратора. Цей клас реалізує частину клієнтського інтерфейсу та функціоналу системи. Він має атрибути, такі як список користувачів, та дані для створення облікового запису. Серед методів представлені такі: стягування списку існуючих користувачів з серверу (та заповнення атрибуту «список користувачів»), створення облікового запису користувача (під час якого заповнюється атрибут «дані для створення облікового запису») та видалення користувачів. Даний клас асоціюється з класом «Адміністратор», так як адміністратор керує панеллю адміністратора.

Клас: Користувач. Даний клас представляє основну інформацію про користувача системи авторизації. Він має атрибути, такі як ім’я, прізвище, ID, нікнейм, адреса електронної пошти, посилання на персональне фото, дата останнього візиту, дата створення користувача, дата оновлення даних користувача. Цей клас містить методи, які дозволяють створювати знімок

обличчя користувача та підтверджувати відправку отриманого зображення на обробку. Цей клас асоційований з класом «Адміністратор», так як обоє класів являються користувачами (узагальнено) системи.

Клас: Сторінка авторизації. Даний клас реалізує частину клієнтського інтерфейсу та функціоналу системи, а саме сторінку авторизації користувача. Він має єдиний атрибут – знімок обличчя у форматі base64. Цей клас має методи, за допомогою яких користувач зможе провести авторизацію (яка включає в себе методи класу «Користувач»), а також метод, який виводить дані користувача після успішної ідентифікації. Цей клас асоційований з класом «Користувач», так як користувач використовує систему авторизації.

Клас: Клієнт (інтерфейс). Цей клас реалізує обгортку класів «Сторінка авторизації» та «Панель адміністратора». Дані класи пов’язані зв’язками композиції і не можуть існувати без класу «Клієнт (інтерфейс)». Цей клас представляє методи обробки запитів та даних, які циркулюють в екземплярах класів, що містяться в класі «Клієнт (інтерфейс)».

Клас: База даних. Даний клас реалізує сутність бази даних, яка містить в собі колекції облікових записів користувачі, а також їхню кількість для відображення на стороні інтерфейсу клієнтської частини. Безпосередньо бере участь в створенні облікового запису користувача.

Клас: Колекція користувачів. Даний клас реалізує інтерфейс класу «Користувач» та має зв’язок композиції з класом «База даних». Таким чином, база даних може мати в собі 0 або безліч користувачів описаних інтерфейсом «Колекція Користувачі», але користувачі описані даним інтерфейсом не можуть існувати без бази даних.

Клас: Amazon S3. Цей клас реалізує сутність хмарного сховища даних. Має єдиний атрибут – зображення. Виконує функцію збереження зображень, даних пов’язаних з розпізнаванням обличчя, а також поверненням цих файлів запитувачу.

Клас: Amazon Rekognition. Цей клас реалізує сервіс, який буде

виконувати розпізнавання облич. Має наступні атрибути: колекція облич – сутність, яка зберігає масив параметрів всіх розпізнаних облич (елемент із колекції можна видобути шляхом запиту до AWS Rekognition по вкладеному атрибуту FaceID) та ідентифікатор користувача (або UserID) – атрибут який означає створених користувачів в процесі розпізнання облич. В процесі реєстрації користувача, UserID прив’язується до розпізнаних облич і зберігається в колекції, для подальшого фільтрованого пошуку. Даний клас має наступні методи: розпізнавання обличчя на зображенні, створення користувача (UserID) при реєстрації нового облікового запису, асоціація розпізнаних облич з користувачем, пошук користувача по розпізнаним обличчям. Має асоціацію з класом «Amazon S3», так як використовує завантажені зображення в розпізнаванні облич і отримує доступ до них методом пошуку файлу по назві, яку передають в одному з вищеописаних методів.

Клас: Сервер (бекенд). Даний клас реалізує серверний додаток, який виступає сервісом для формування запитів до класів «Amazon Rekognition», «Amazon S3», «База даних», а також обробником отриманих результатів запитів. Вищезазначені класи мають зв’язок агрегації з поточним класом, база даних і сервіси AWS можуть існувати і функціонувати без бекенд-додатку. Цей клас асоційований з класом «Клієнт (інтерфейс)» і виступає в ролі частини цілого. Даний клас може існувати в разі відсутності клієнту. Методами класу виступають: збереження та повернення облікових записів користувачів, завантаження зображень до хмарного сховища S3, конструювання запитів до класу «Amazon Rekognition», виконання запитів та обробка відповіді.

Таким чином, ми отримуємо повністю описані класи системи авторизації, які демонструють майбутній функціонал додатку в повному обсязі. Окрім того, ми описали взаємозв’язки між класами, серед яких були асоціації, агрегації та композиції. Кожен вид зв’язку показує в той чи іншій мірі важливість та ранг класу відносно інших. Описавши сутності, їх атрибути та поведінку, на основі цієї інформації ми можемо побудувати діаграму класів (див. рис. 2.1).

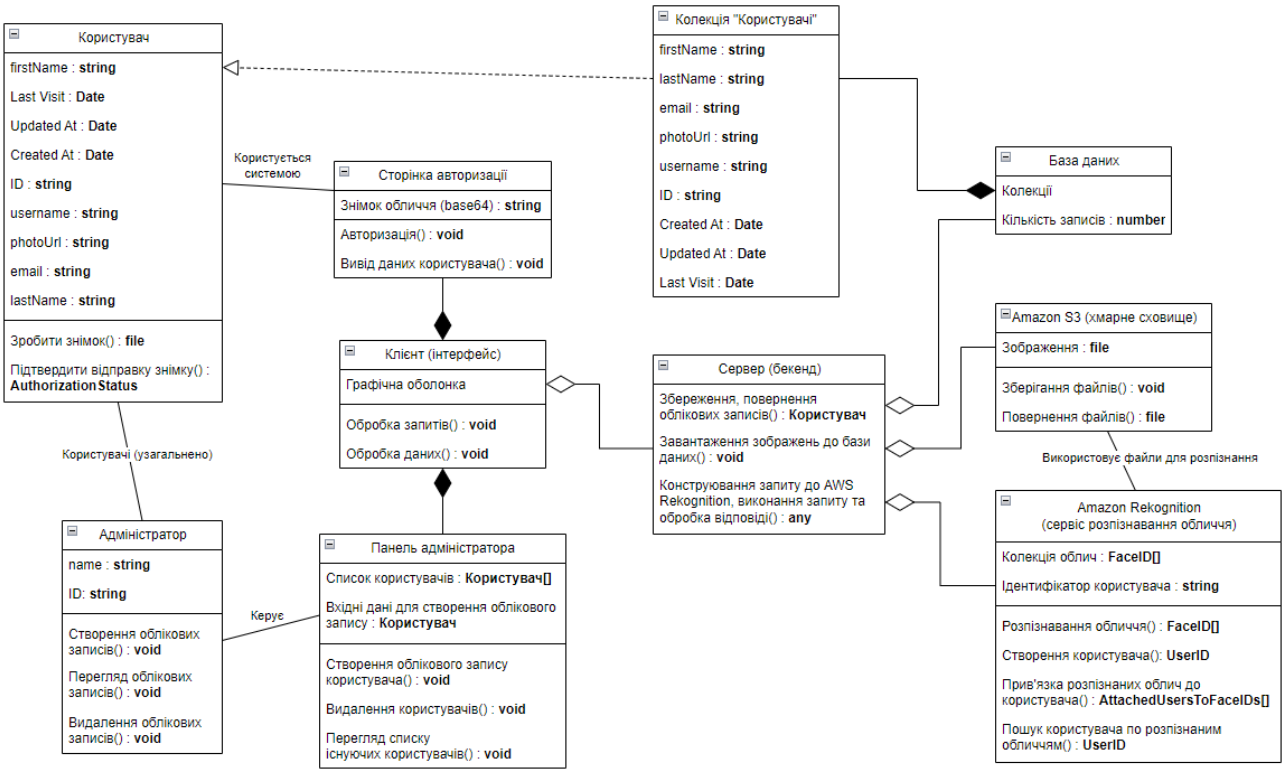

Рисунок 2.1 – Діаграма класів системи

На представленій діаграмі класів системи, відображено саме ті класи, які були перелічені та охарактеризовані вище.

Виходячи з відображеної інформації в даній діаграмі, на цьому етапі ми вже можемо уявити структуру системи та її складову, що є дуже важливим аспектом під час проєктування системи.

2.2 Діаграма послідовності

Діаграма послідовності – це вид UML-діаграми, який моделює взаємодії між об’єктами в системі у вигляді послідовності повідомлень. Вона використовується для візуалізації порядку обміну повідомленнями між різними елементами системи, такими як об’єкти або класи, та показує послідовність цих взаємодій у часі.

Об’єкти системи розташовуються на вертикальних лініях, відомих як лінії життєвого циклу, а взаємодії між об’єктами показані стрілками.

2.2.1 Діаграма послідовності створення користувача

Для початку, давайте переглянемо сконструйовану діаграму послідовності, яка відображає функціонал створення користувача через інтерфейс панелі адміністратора (див. рис. 2.2 та 2.3).

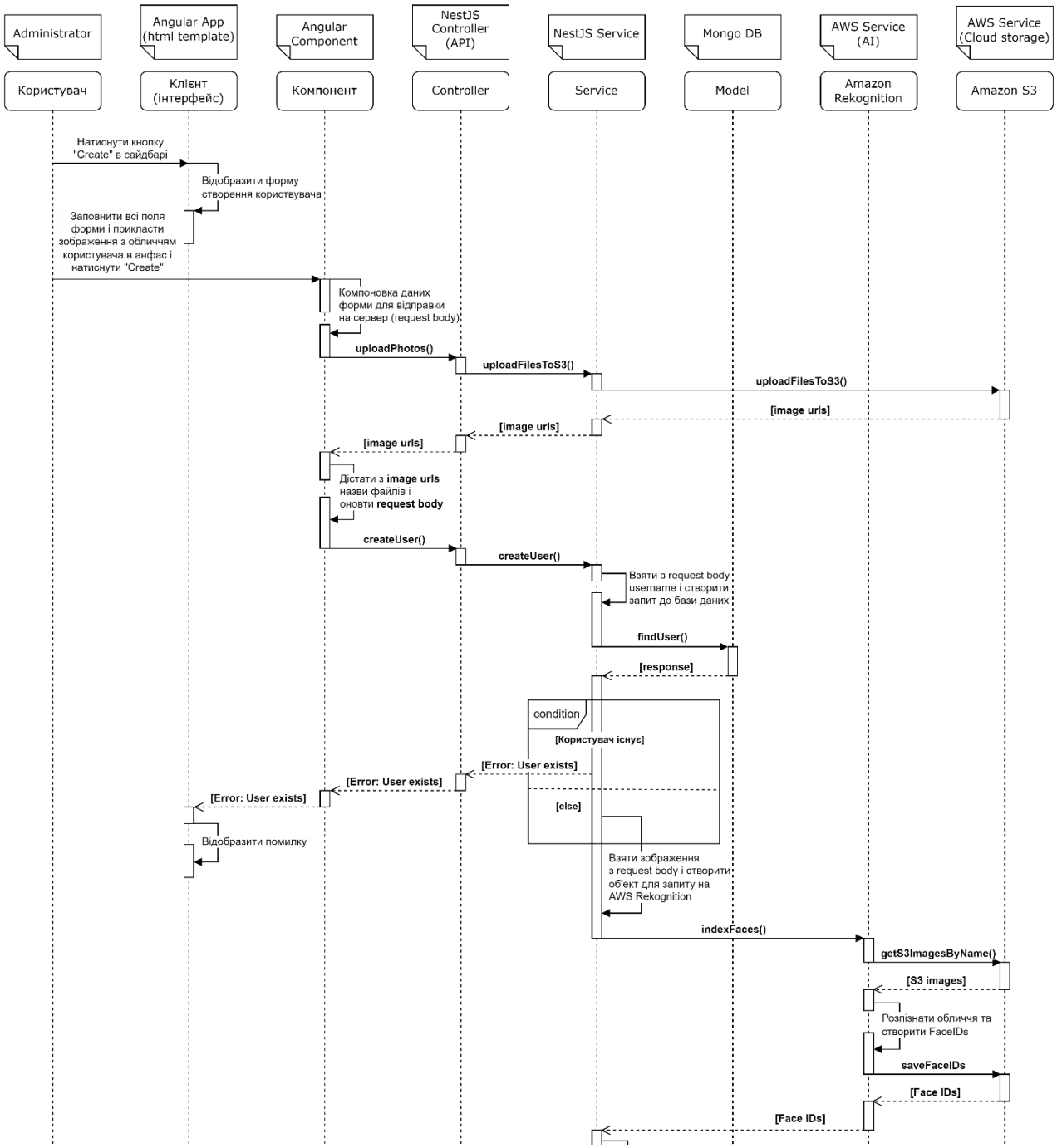

Рисунок 2.2 – Діаграма послідовності для функціоналу «Реєстрація користувача»

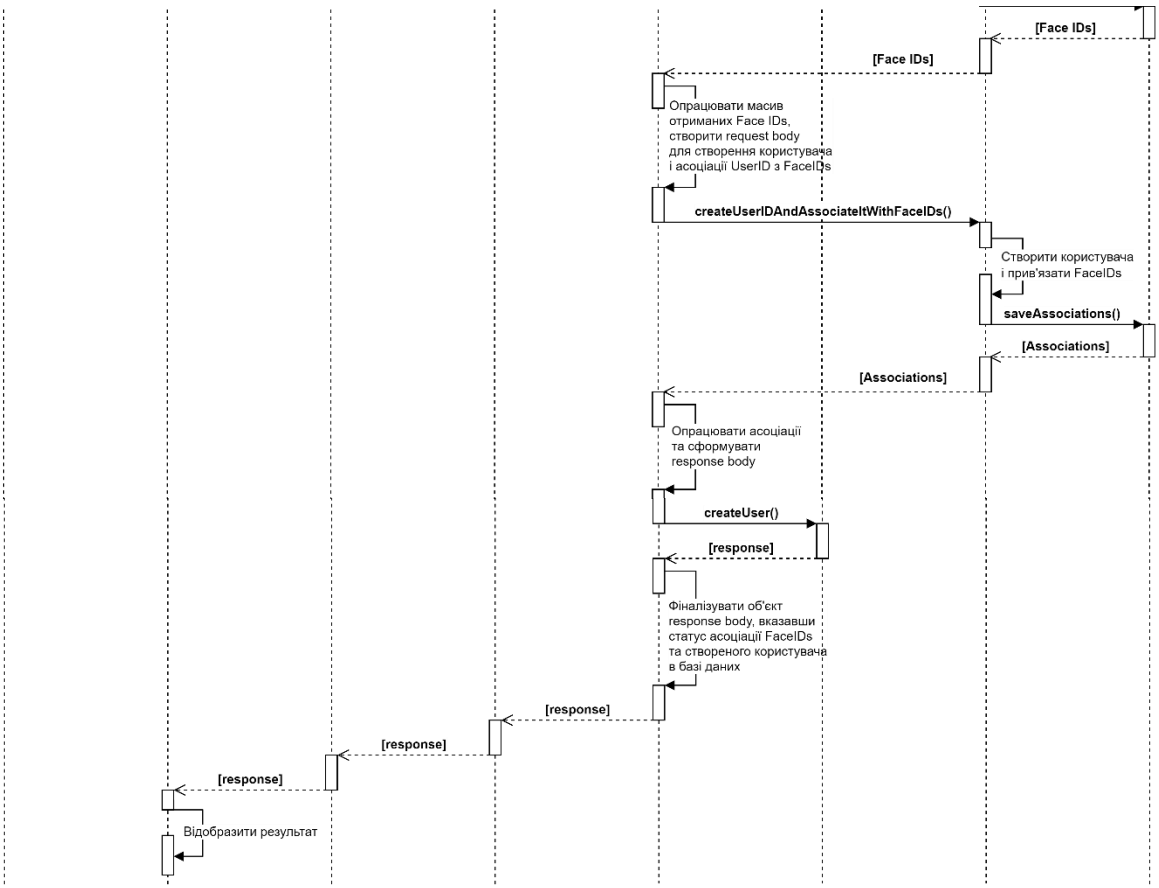

Рисунок 2.3 – Діаграма послідовності для функціоналу «Реєстрація користувача»

Дана діаграма являється спрощеним варіантом, так як деталізована версія може зайняти дуже багато місця, а велика кількість елементів може ускладнити читабельність головного потоку, який демонструє функціонал цієї частини системи. Діаграма, яку ми щойно переглянули, демонструє поведінку створення користувача, якою керує адміністратор системи.

Перш за все, коли адміністратор потрапляє до інтерфейсу нашого додатку, він повинен обрати сторінку, яку він хоче відвідати. Якщо ми говоримо про створення користувача, то він повинен натиснути на відповідну кнопку – «Create», після цього адміністратор опиниться на потрібній вкладці.

Адміністратора зустріне форма з чотирма полями: нікнейм, ім’я, прізвище та електронна пошта. Також є кнопка для завантаження власного зображення користувача. Праворуч від форми буде спеціальний блок для завантаження декількох зображень, які будуть використані під час створення облікового запису

користувача, для ідентифікації облич користувача, а також в подальшому використанні цих даних безпосередньо під час авторизації користувача.

Припустивши існування таких елементів інтерфейсу майбутньої системи та їх розташування на сторінці, ми можемо перейти до функціональної частини даного потоку.

Спочатку, адміністратор заповнює відповідні поля інформацією, яка відповідає деяким персональним даним користувача котрого бажають зареєструвати в системі. Після цього, він завантажує зображення, які містять обличчя майбутнього користувача в анфас, у відведений блок поряд з формою. Адміністратор натискає кнопку «Create», введені дані формують об’єкт, який буде використаний у створенні облікового запису, але перед створенням користувача – додаток зберігає завантажені зображення в хмарному сховищі Amazon S3, так як сервіс, який буде виконувати розпізнання облич з фотографій, буде звертатись до попередньо завантажених файлів у хмарне сховище.

Слідом за цим, компонент нашого додатку звертається до бекенду, передаючи йому об’єкт з інформацією користувача. Контролер бекенд-додатку перехоплює HTTP-запит і направляє його до сервісу, де першим чином сам сервіс звертається до бази даних MongoDB, перевіряючи, чи не існує користувача з такими ж даними, які передав адміністратор.

Унікальним полем в об’єкті користувача являється поле username, тому всі подальші операції пов’язані з пошуком, створенням або оновленням будуть прив’язані до вищезазначеного поля. Повертаючись до перевірки існуючого користувача, після отримання результатів пошуку, ми робимо перевірку: якщо користувач вже існує, ми повертаємо помилку до клієнту системи, якщо ні – ми переходимо до наступного кроку.

В тілі методу, наступне після запиту до бази даних, йде компонування об’єкту-запиту до сервісу AWS Rekognition. Під час запиту, ми передаємо назву завантажених файлів до хмарного сховища, штучний інтелект, отримавши ці файли, розпізнає обличчя, характеризує їх та присвоює ідентифікатор FaceID.

Після цього, він повертає масив індексованих облич, який ми форматуємо у надлежний для використання вигляд.

Отримавши індексовані обличчя, сервіс робить запит до AWS Rekognition в цілях створення сутності користувача, котрий в результаті повертає нам UserID (цій змінній було присвоєно значення поля username). Далі, ми робимо запит для створення асоціації UserID з масивом облич FaceID. Наприкінці, після успішної асоціації, ми створюємо користувача безпосередньо в базі даних MongoDB, зберігаючи дані, які були надані клієнтом системи та повертаємо результат виконання адміністратору.

Вищеописаний процес досконало характеризує поведінку системи під час створення облікового запису. Побудована діаграма додатково ілюструє потік і взаємодію компонентів, задля поглибленого уявлення про функціональну частину системи.

2.2.2 Діаграма послідовності авторизації користувача

Ознайомившись з діаграмою послідовності створення облікового запису, перейдемо до опису потоку авторизації користувача та побудови діаграми послідовності.

Даний потік буде дещо схожим з послідовністю описаною для функціоналу створення користувача, але меншим по розміру і з меншою кількістю запитів до сервісів AWS, так як функція авторизації використовує вже створені сутності на етапі реєстрації користувача адміністратором системи.

Розпочинається все з того, що користувач потрапляє на сторінку авторизації, і його зустрічає модальне вікно, яке містить якусь інформацію для ознайомлення з сервісом і кнопку «Sign In».

Після того як користувач натискає на кнопку, інтерфейс модального вікна змінює свій зовнішній вигляд, де з’явиться вікно, яке буде транслювати дані з

камери, яку користувач попередньо ввімкне (про це браузер запитає користувача). Також тут буде кнопка, яка буде називатись «Take a Photo», після натискання на яку, програма зафіксує знімок з камери і відразу відобразить цей знімок на місці, де транслювалось відео з камери. Компонент за допомогою HTTP-запиту зберігає це зображення в хмарному сховищі, і після цього звертається до бекенд-додатку вказуючи ім’я завантаженого зображення.

На стороні бекенд-додатку, сервіс робить запит до AWS Rekognition під назвою «SearchUsersByImage», штучний інтелект ідентифікує обличчя з попередньо завантаженого зображення до S3 та порівнює отримані параметри з вже існуючими FaceID, після цього повертає результати ідентифікації до бекенд-сервісу.

Сервіс перевіряє чи масив ідентифікованих облич не пустий, потім перевіряє перший елемент масиву (за документацією, AWS Rekognition повертає масив об’єктів, де перший елемент має найбільший рівень співпадіння), якщо поле «Similarity» має значення більше за 96 відсотків (або будь-яке інше значення, яке буде виступати порогом пропуску значення схожості облич і може бути виставлене на розсуд розробника), то сервіс бере прив’язаний UserID (який був прив’язаний на етапі асоціації FaceIDs та UserID під час створення облікового запису), і робить пошук користувача у базі даних MongoDB по полю username, зі значенням поля UserID (так як ці значення були пов’язані на етапі створення облікового запису) і повертає результат пошуку, тобто сутність «Користувач» до клієнту системи, який в свою чергу відображає всю інформацію про користувача в інтерфейсі додатку.

Якщо сервіс розпізнання не знаходить співпадінь при порівнянні ідентифікованих облич, цей випадок обробляється бекенд-сервісом і результат безуспішної авторизації повертається до клієнту системи, де відображається відповідний інтерфейс.

Ознайомившись з описом потоку, переглянемо побудовану діаграму послідовності авторизації користувача на рисунку 2.4.

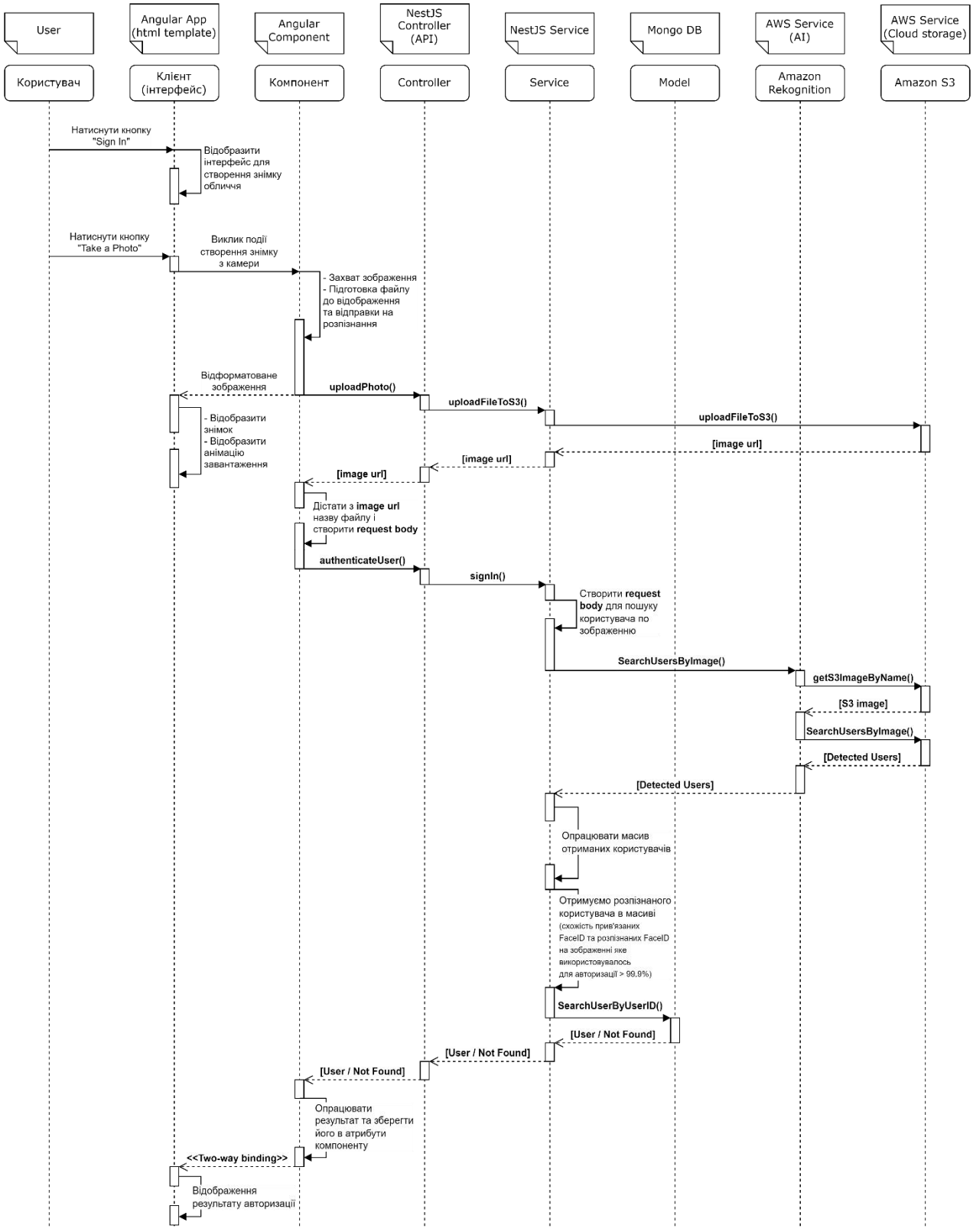

Рисунок 2.4 – Діаграма послідовності авторизації користувача

Ця діаграма послідовності детально описує потік функціонала авторизації, зображує об’єкти, які беруть участь в потоці та їх взаємодію між собою.

2.3 Дизайн системи

Дизайн – це один із обов’язкових етапів проєктування будь-якого програмного забезпечення. Цей етап розробки охоплює різні аспекти, такі як структура бази даних, вибір архітектури програми, розташування функцій та об’єктів, прийоми взаємодії між компонентами та інші аспекти, необхідні для створення працездатної системи.

Основною метою дизайну є створення гнучкого, ефективного та масштабованого програмного додатку, який задовольнить вимоги користувачів та вирішить поставлені завдання.

У процесі розробки програмного забезпечення дизайн відіграє важливу роль, оскільки він визначає базову архітектуру, диктує поведінку системи в її різних станах та описує структуру програми перед початком самого процесу розробки.

Якість дизайну та увага дизайнера до деталей на цьому дизайні впливає на подальшу роботу над проєктом та його ефективність у вирішенні поставлених завдань.

Отже, побудувавши діаграми класів та послідовності, ми можемо перейти до етапу створення дизайну додатку, взявши за основу зібрану та добре структуровану інформацію з вищезгаданих діаграм. Для створення дизайну ми будемо використовувати онлайн-сервіс для розробки інтерфейсів Figma [5].

2.3.1 Дизайн панелі адміністратора

Панель адміністратора в нашому додатку буде складатись із двох сторінок, сторінки створення користувача та сторінки перегляду таблиці з існуючими користувачами. Давайте переглянемо дизайн цих двох сторінок, зображених на рисунках 2.5 та 2.6.

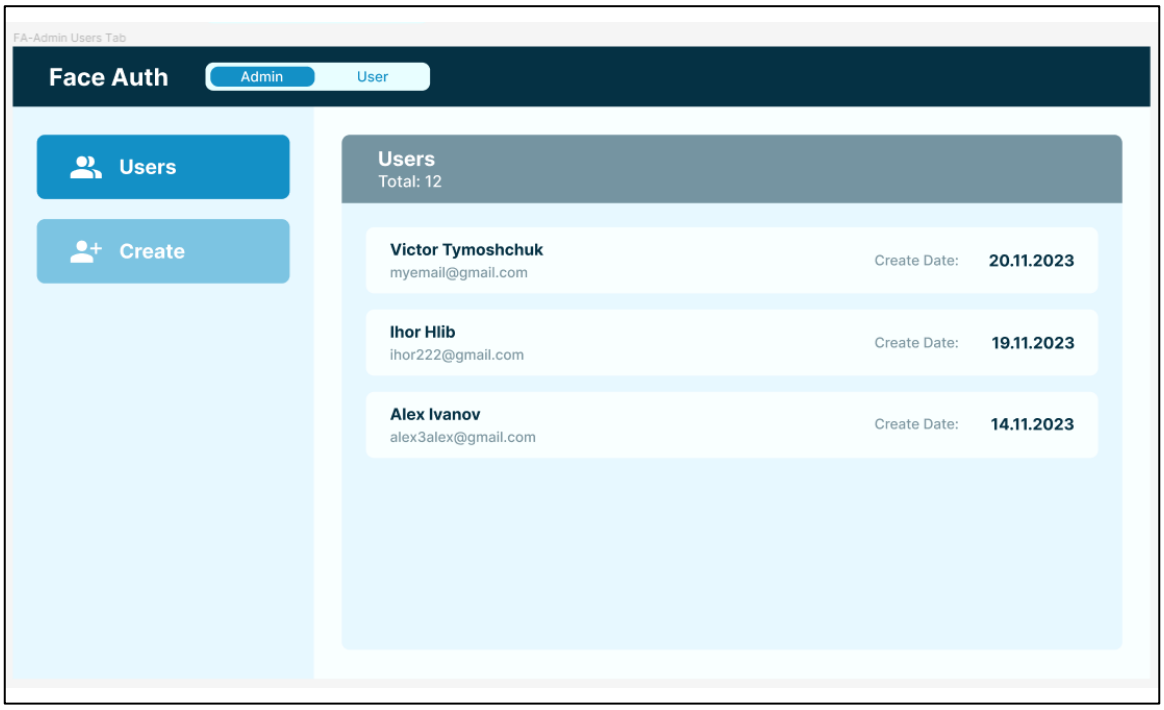

Рисунок 2.5 – Дизайн сторінки з таблицею користувачів

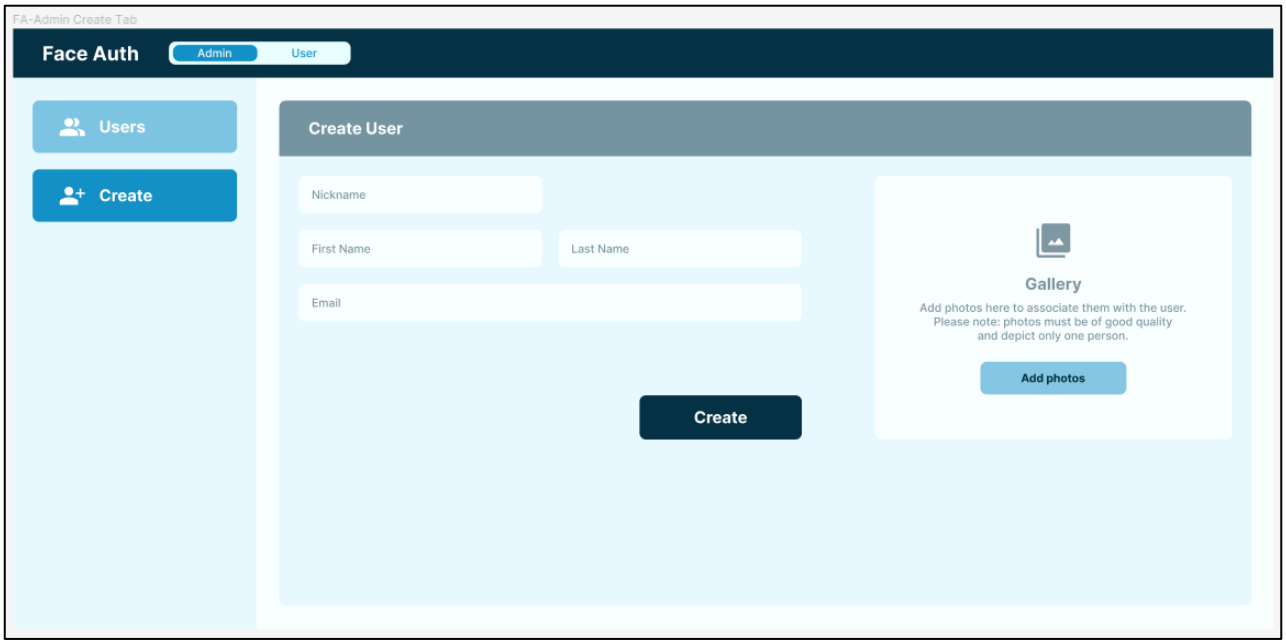

Рисунок 2.6 – Дизайн сторінки створення користувача

Даний дизайн сповна ілюструє майбутній вигляд системи авторизації з боку панелі адміністратора, де сам адміністратор зможе додавати користувачів і переглядати їх список, користуючись інтерфейсом, який буде описаний цим дизайном.

2.3.2 Дизайн сторінки авторизації

Далі, перейдемо до дизайну сторінки авторизації, якою буде користуватись потенційний клієнт системи (див. рис. 2.7). Дизайн цієї сторінки буде складатись із трьох частин, які будуть демонструвати три кроки процесу авторизації відповідно.

Перший крок буде інформативного та косметичного характеру, другий крок буде функціональним і відігравати головну роль в процесі авторизації, на даному етапі користувач захопить знімок з камери пристрою та ініціює процес авторизації, третій крок буде результуючим – після успішної авторизації, користувач буде переходити до цього кроку і зможе розглянути результат авторизації, де будуть зображені персональні дані користувача, які адміністратор вказав на етапі реєстрації облікового запису.

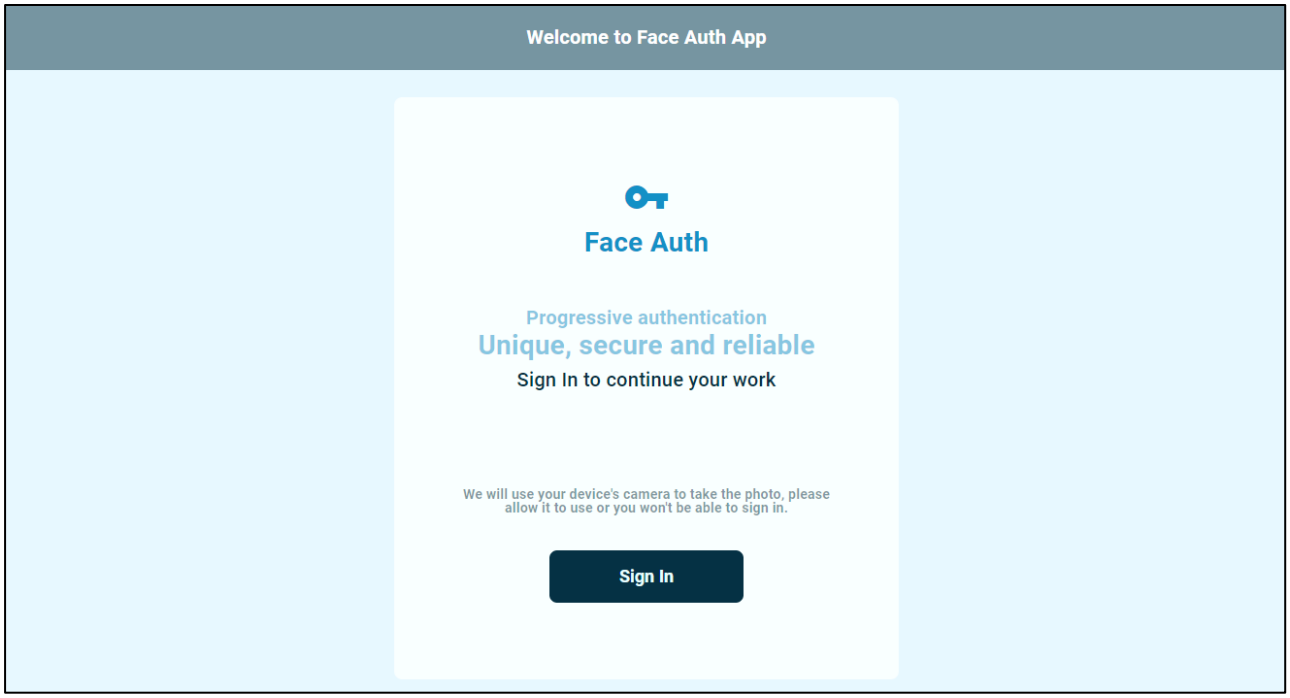

Рисунок 2.7 – Дизайн сторінки авторизації (крок: 1)

На зображенні, що ми щойно переглянули, відображено перший крок модального вікна, який ознайомлює користувача з системою і пропонує користувачеві авторизуватись.

Після натискання кнопки «Sign In», він перейде до другого кроку процесу авторизації (див. рис. 2.8).

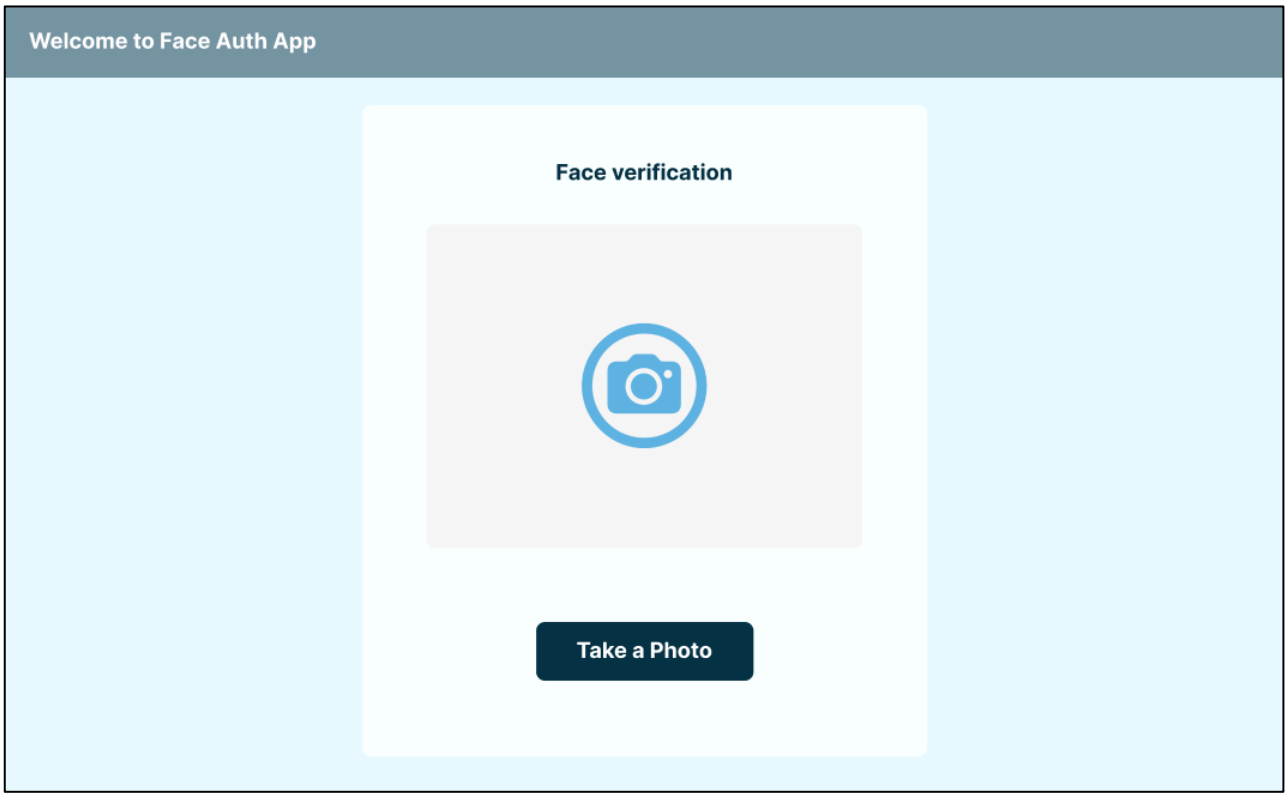

Рисунок 2.8 – Дизайн сторінки авторизації (крок: 2)

На даному етапі, браузер запитає користувача про дозвіл використання камери девайсу, після отримання дозволу, область з іконкою камери почне транслювати відео з камери. Окрім того, функціонал підтримує трансляцію відео з телефону, можна використовувати фронтальну камеру або перемкнутись на основну.

Після натискання на кнопку «Take a Photo», ми відобразимо іконку-спінер, яка буде показувати статус завантаження. Одночасно з цим розпочнеться сам процес авторизації, який був описаний у попередніх пунктах.

Після успішної авторизації користувач переходить до третього і останнього кроку авторизації. На ньому ми відображаємо картку користувача, яка містить поля з іменем, прізвищем, нікнеймом та адресом електронної пошти. Переглянемо дизайн останнього кроку авторизації (див. рис. 2.9).

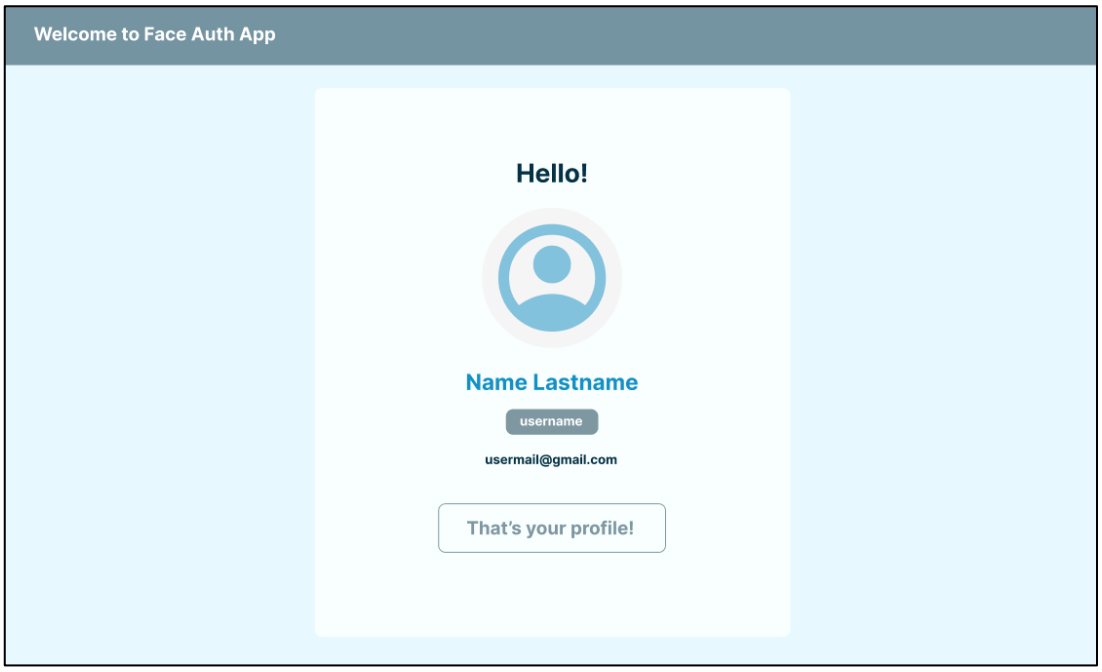

Рисунок 2.9 – Дизайн сторінки авторизації (крок: 3)

В результаті проєктування візуальної частини додатку, ми отримали дизайн, який ілюструє всі можливі функціональні сторінки, які будуть використовуватись для створення, перегляду та редагування користувачів, а також авторизації цих користувачів у системі.

В цілому, процес проєктування системи можна рахувати завершеним, ми маємо достатньо інформаційних ресурсів, щоб розпочати наступний етап розробки – реалізацію додатку.

3 РЕАЛІЗАЦІЯ СИСТЕМИ АВТОРИЗАЦІЇ

3.1 Опис інструментів розробки

Для реалізації додатку були використані фронтенд фреймворк Angular та бекенд фреймворк NestJS.

RxJs – бібліотека створена для реактивного програмування на мові JavaScript [6]. Являється вбудованою в Angular CLI, після створеного проєкту-шаблону, RxJs можна використовувати відразу без додаткових встановлень і налаштувань.

Material Design – найпопулярніша дизайн-система для Angular фреймворку, розроблена компанією Google [7]. Просте встановлення, просто використовувати, виглядає самодостатньо і досить непогано вписується в мінімалістичний дизайн.

Tailwind CSS – це CSS-фреймворк, який реалізує набір готових стилів та класів для швидкої розробки та стилізації компонентів. За допомогою нього, шляхом вводу спеціальних класів прямо в HTML шаблоні можна стилізувати елементи без написання CSS у відведеному файлі.

MongoDB – база даних, яку використовує система для збереження сутностей користувачів. Була обрана через просте впровадження в бекенд-додаток, пакет «nestjs/mongoose» реалізує зручну утиліту для взаємодії з базою даних, має обширну документацію і простий синтаксис.

AWS Rekognition – сервіс з потужним штучним інтелектом розпізнання облич, тексту та інших об’єктів. Також має встановлювані пакети, які зводять складність роботи з сервісом до мінімуму.

Amazon S3 – хмарне сховище, використовується для зберігання зображень, які будуть підлягати розпізнаванню, а також являються зручним сервісом для зберігання файлів.

3.2 Реалізація фронтенд частини

Щоб почати роботу над клієнтською частиною нашої системи, потрібно створити Angular проєкт за допомогою Angular CLI [8]. Це робиться методом вводу команди «ng new project-name» в командний рядок. Після успішного створення проєкту, ми можемо приступати до розробки.

Для того щоб архітектура додатку була масштабованою і більш зрозумілою, ми впровадимо систему роутинга. Створимо два нових модуля та компоненти ввівши команди «ng g m» та «ng g c», а в кореневому компоненті створимо два шляхи: user та admin. Кожен шлях прив'яжемо до попередньо створених модулів та компонентів (див. рис. 3.1).

src > app > app-routing.module.ts > ...

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3  import { UserSignInComponent } from './user-sign-in/user-sign-in.component';
4
5  const routes: Routes = [
6    {
7      path: '',
8      pathMatch: 'full',
9      redirectTo: 'admin',
10   },
11   {
12     path: 'admin',
13     loadChildren: () => import('./admin/admin.module').then((m) => m.AdminModule),
14   },
15   {
16     path: 'user',
17     component: UserSignInComponent
18   },
19 ];
20
21 @NgModule({
22   imports: [RouterModule.forRoot(routes)],
23   exports: [RouterModule]
24 })
25 export class AppRoutingModule { }
26
```

Рисунок 3.1 – Роутинг додатку

Таким чином, код зображений на переглянутому рисунку реалізує систему роутингу. Після того, як користувач потрапляє до кореневого шляху додатку, він буде автоматично переадресований до роуту admin.

З роутингом закінчили, перейдемо до реалізації коду в модулі admin. Першим чином створимо два компонента, де один з них буде відповідати за реалізацію сторінки створення користувача, а інший – за відображення існуючих користувачів. Ось так виглядає архітектура проєкту після всіх вищезазначених маніпуляцій (див. рис. 3.2).

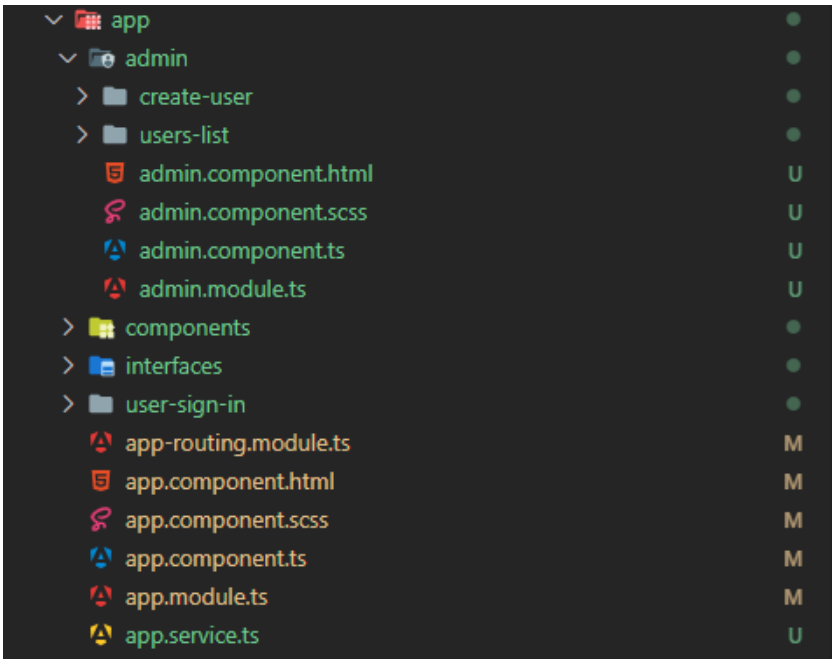

Рисунок 3.2 – Архітектура проєкту

Прикладом реалізації надалі послугує інтерфейс створення користувача. Отже, у модулі реєстрації користувача імпортуємо модулі FormsModule та ReactiveFormsModule, які дозволять нам працювати з формами у реактивному стані, швидко отримуючи доступ до користувацького вводу в компоненті модулю. Створимо сутність форми, яка містить в собі декілька FormControl, котрі ми прив’яжемо до полів вводу в HTML-темплейті (див. рис. 3.3, 3.4).

```
23  fg = new FormGroup({
24    username: new FormControl('', Validators.required),
25    firstName: new FormControl('', Validators.required),
26    lastName: new FormControl('', Validators.required),
27    email: new FormControl('', Validators.required),
28  })
```

Рисунок 3.3 – Код ініціюючий сутність форми та її полів

```
<form [formGroup]="fg" class="w-full h-max grid grid-cols-2 gap-5 max-w-[520px]">
  <input class="fa-input w-full col-span-1" type="text" formControlName="username" placeholder="Unique Username">
  <div class="w-full flex gap-5 col-span-2">
    <input class="fa-input w-full" type="text" formControlName="firstName" placeholder="First Name">
    <input class="fa-input w-full" type="text" formControlName="lastName" placeholder="Last Name">
  </div>
  <input class="fa-input w-full col-span-2" type="text" formControlName="email" placeholder="Email">
  <div class="w-[120px] h-[120px] flex relative user-photo-wrapper overflow-hidden">
    <input type="file" name="img-input" id="photoSelect" (change)="onPhotoSelected($event)">
    <img *ngIf="img" [src]="img" class="w-full h-full object-cover z-[2]" alt="User Image">
    <mat-icon class="!text-[40px] !w-[40px] !h-auto absolute top-[calc(50%-20px)] left-[calc(50%-20px)] z-[1]">photo_camera</mat-icon>
  </div>
  <div class="w-full flex justify-end mt-[70px] col-span-1">
    <button *ngIf="!loading" [disabled]="imgLoading" class="fa-primary-button w-[200px] !h-[54px]" (click)="createUser()">CREATE</button>
    <mat-progress-spinner *ngIf="loading" [color]="'primary'" [mode]="'indeterminate'" [diameter]="46"></mat-progress-spinner>
  </div>
</form>
```

Рисунок 3.4 – Код HTML-темплейту з демонстрацією прив’язки полів до сутності форми створеної в компоненті

Як можна помітити, до тегу form ми додаємо декоратор [formGroup] і присвоюємо йому атрибут форми, який ми ініціювали в компоненті [9]. Кожному input ми додаємо formControlName, і зв’язуємо поля вводу до кожного FormControl (ініційовані в компоненті) методом вводу назви поля, таким чином реалізуючи концепцію двонапрямкового зв’язування даних – Two-way binding [10]. Також, на цьому зображенні ми можемо переглянути як реалізується стилізація за допомогою Tailwind CSS.

Тепер можемо перейти до написання коду компоненту, який буде завантажувати зображення до S3, та комунікувати з бекенд-додатком. Код завантаження зображення до хмарного сховища являється дещо ускладненим, переглянути його ви зможете у Додатку А.

Принцип завантаження виглядає наступним чином: ми звертаємось до бекенд-додатку з метаданими зображення, бекенд-додаток за допомогою одного з AWS сервісів створює «підписане» посилання, яке повертає до клієнту системи. На стороні клієнту ми беремо це посиланням і за допомогою HTTP методу PUT, завантажуємо зображення до Amazon S3. Таким чином, ми створюємо посилання для завантаження файлу, яке диктує правила завантаження та захищає хмарне сховище від вхідних файлів з невідомих джерел, проводить валідацію завантажуваного файлу.

Наступним кроком є імплементація функції, яка буде компонувати об’єкт з введеними в форму даними користувача та відправляти їх до бекенд-додатку

задля створення облікового запису. Реалізація цієї функції виглядає наступним чином (див. рис. 3.5).

```
createUser() {
  if (this.fg.invalid) return;
  if (!this.imgToRekognize.length) return;
  this.loading = true;
  const body: any = this.fg.value;
  body.images = this.imgToRekognize;
  body.userPhoto = this.img || null;
  this.appService.createUser(body).pipe(
    catchError((err: HttpErrorResponse) => {
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: any) => {
    console.log(data);
    this.openSnackBar('Successfully created!');
    this.loading = false;
  })
}
```

Рисунок 3.5 – Код функції компоновки об’єкту для створення користувача

Дана функція перед виконанням перевіряє чи всі поля форми заповнені, чи завантажене хоч одне зображення для ідентифікації облич створюваного користувача, потім створюється request body і передається в сервіс appService, який виконує POST HTTP запит до бекенд додатку [11]. Тут для обробки HTTP запитів ми використовуємо RxJs: catchError виконує функцію перехоплювача помилок, subscribe створює підписку на потік даних. Підписка на потік даних ініціює HTTP запит. В результаті ми показуємо снекбар (блок-нотифікація) про статус виконання запиту (див. рис. 3.6) і припиняємо завантаження (яке відображється в HTML-темплейті в вигляді іконки-спінера).

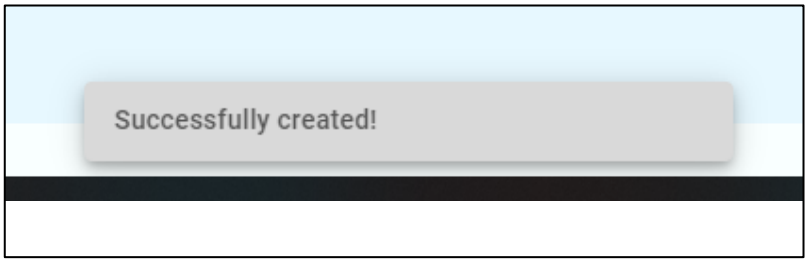

Рисунок 3.6 – Вигляд блоку з нотифікаціями

Результатом написання коду, який буде виконувати функцію створення користувача являється наступна повнофункціональна сторінка (див. рис. 3.7).

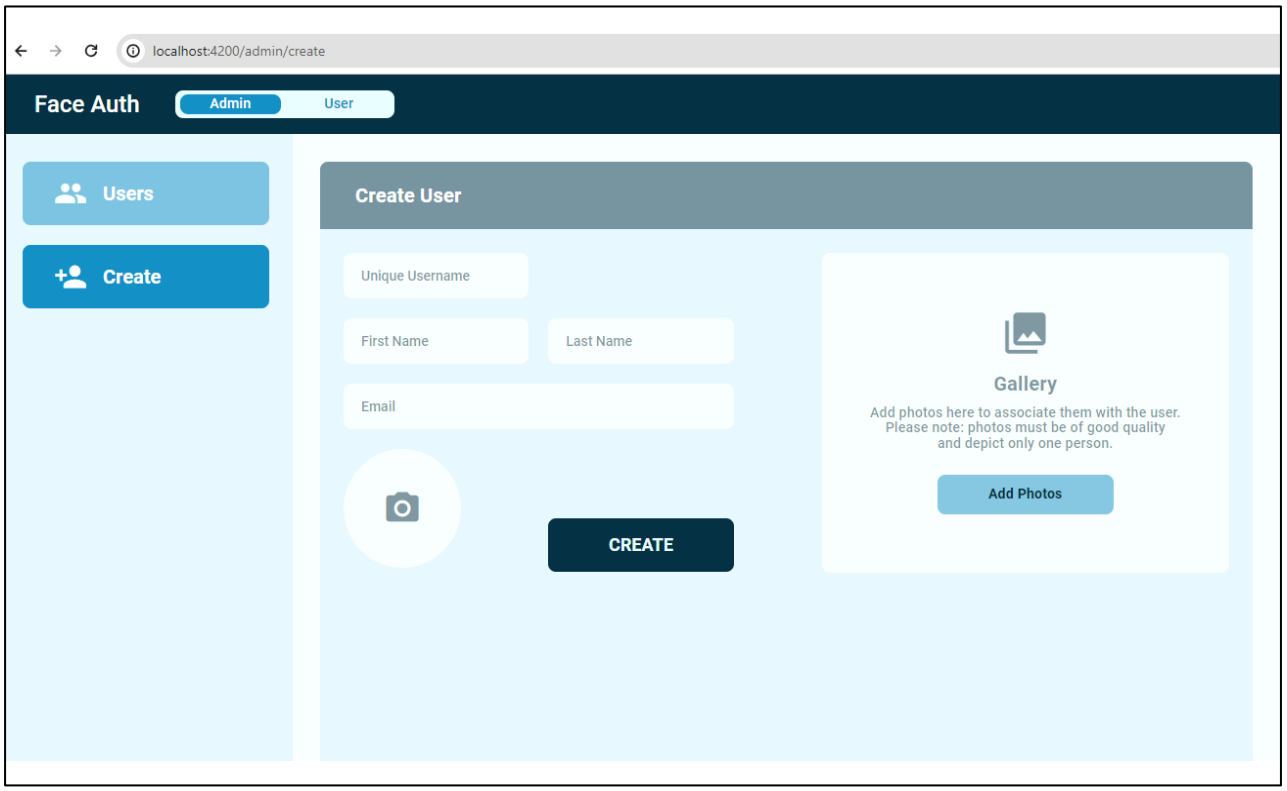

Рисунок 3.7 – Фінальний вигляд сторінки створення користувача

Як бачимо, реалізована сторінка має всі необхідні поля та блоки для завантаження зображень та послідуючого створення облікового запису користувача.

Важливо додати, що сторінки були реалізовані з деякими візуальними поліпшеннями, щоб підвищити інформативність системи та покращити якість користувацького досвіду у використанні додатку. Наприклад, на сторінці створення користувача було додано метод вводу персонального зображення, на сторінці відображення таблиці користувачів, додано вивід цього ж зображення, а також поле, яке позначує дату останнього візиту в систему. Інтерфейс сторінок авторизації та перегляду таблиці користувачів були імплементовані у схожий спосіб до реєстрації клієнтів. Результат реалізації сторінки списку користувачів можна переглянути на рисунку 3.8.

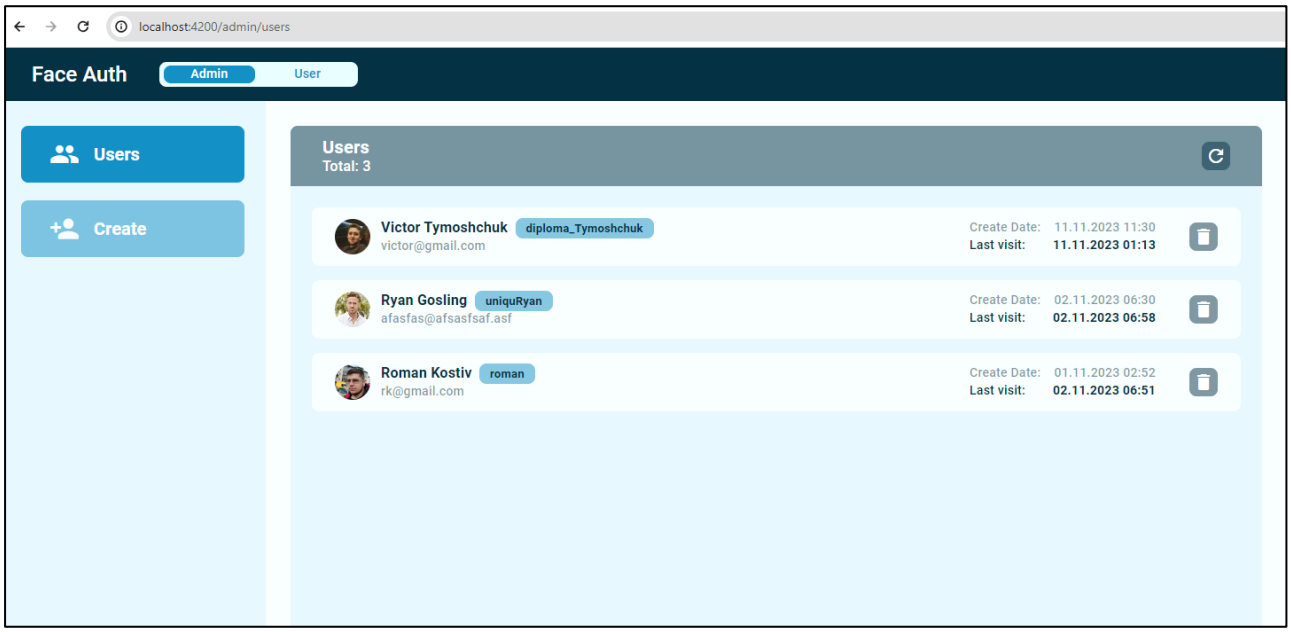

Рисунок 3.8 – Реалізація сторінки таблиці користувачів

Реалізацію сторінки авторизації користувача зображено на рисунку 3.9.

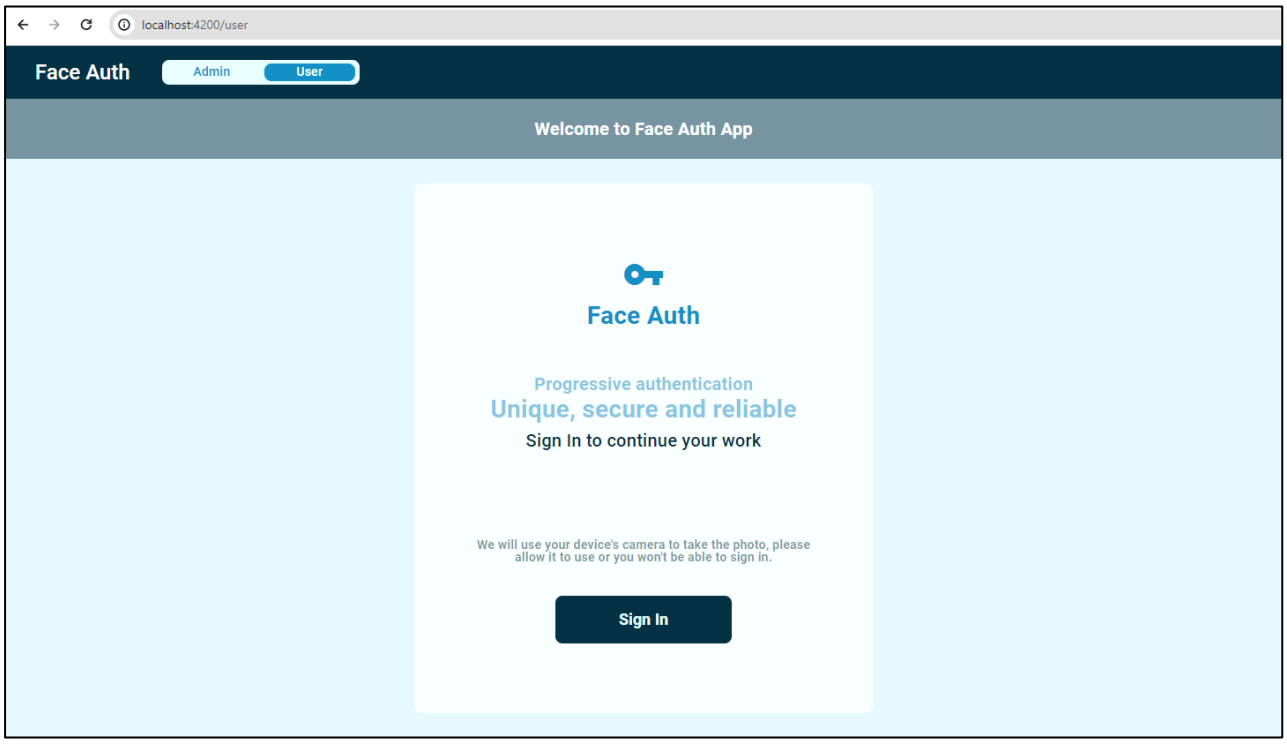

Рисунок 3.9 – Реалізація сторінки авторизації

На цьому реалізацію клієнтської частини системи завершено, в результаті розробки було створено інтерфейси сторінки авторизації та її 3 етапи

відображення, сторінки створення користувача та сторінки відображення таблиці користувачів. Повний код цих модулів можна переглянути у додатку Б. Наступним кроком буде реалізація бекенд-додатку з використанням NestJS.

3.3 Реалізація бекенд частини

Для початку роботи з бекенд-додатком, нам потрібно створити новий NestJS проєкт, ввівши команду «nest new project-name» в командний рядок, ідентично до того, як ми це робили з фреймворком Angular.

Перед початком розробки коду, потрібно встановити наступні пакети для роботи з базою даних та сервісами AWS:

– npm i --save class-validator class-transformer;
– npm i --save @nestjs/config;
– npm i --save @nestjs/mongoose mongoose;
– npm i --save @aws-sdk/client-rekognition;
– npm i --save @aws-sdk/s3-request-presigner.

Перші три пакета встановлюють утиліту для роботи з базою даних MongoDB, надають можливість працювати з .env файлом (файл який містить в собі змінні середовища), та клас-валідатор, який допоможе валідувати вхідні дані в середині контролеру бекенд-додатку.

Два останні пакети являються утилітами для роботи з AWS сервісами. Пакет «s3-request-presigner» дає можливість зарезервувати місце на хмарному сховищі, створюючи підписане посилання, по якому ми на клієнтській частині додатку завантажуємо зображення. Пакет «client-rekognition» дозволяє максимально просто створювати запити до AWS Rekognizer в два кроки, створивши об’єкт запиту, який трактується уніфікованим шаблоном та інтерфейсами і безпосередньо виконання запиту через вбудований метод «send()».

Встановивши всі необхідні пакети, запускаємо наш додаток ввівши

команду «npm run start:dev» та переходимо до написання коду.

Першим чином, в кореневому модулі додатку імпортуємо модуль ConfigModule (для доступу до змінних з .env файлу) та MongooseModule, в якому ми вкажемо посилання до нашої бази даних, за допомогою якого наш бекенд-додаток буде проводити підключення.

Після цього, створюємо два модулі: модуль для роботи з завантаженням зображень і модуль для роботи з AWS Rekognition. Це робиться ідентично до того, як ми робили це з фреймворком Angular, вводимо команди «nest g m», «nest g s» (сервіс) та «nest g c» (контролер).

Давайте розглянемо імплементацію потоку створення користувача, так як воно дуже обширне і найкраще описує процес розробки і всі тонкощі розробки бекенд-додатку. Для початку, переглянемо код контролеру модулю «aws», який перехоплює запити по роуту «/aws/create-user» за допомогою функції createUser з декоратором «@Post», який явно вказує яким HTTP методом звертатись до даного роуту (див. рис. 3.10).

```
4   @Controller('aws')
5   export class AwsController {
6
7       constructor(private readonly awsService: AwsService) {}
8
9       @Post('create-user')
10      async createUser(@Body() body: {
11          username: string;
12          firstName: string;
13          lastName: string;
14          userPhoto: string | null;
15          email: string,
16          lastVisit: string;
17          images: string[]
18      }) {
19          return await this.awsService.createUser(body);
20      }
```

Рисунок 3.10 – Код контролеру модуля «aws»

Після перехоплення запиту з вказаними полями в тілі об’єкту, ми перенаправляємо їх в наш сервіс, який і виконує головну роль у створенні користувача. Далі ми розглянемо як був реалізований сервіс для створення

облікового запису користувача, розіб'ємо його на декілька блоків та детально розглянемо послідовність дій, результатом яких являється створений користувач у базі даних та ідентифіковані обличчя цього користувача, які будуть використані під час авторизації. Переглянемо першу частину коду на вході у функцію createUser (див. рис. 3.11).

```
204     async createUser(userData: {
205         username: string;
206         firstName: string;
207         lastName: string;
208         userPhoto: string | null;
209         email: string;
210         images: string[];
211     }) {
212         const user = await this.userModel.findOne({
213             username: userData.username,
214         });
215         if (user) {
216             throw new UnauthorizedException('User already exists');
217         }
```

Рисунок 3.11 – Код перевірки на існування користувача

На попередньому рисунку зображено код, який першим чином дістає поле username з request body і шукає користувача в базі даних по цьому полю, так як воно у нас виступає унікальним ідентифікатором. У разі, якщо в результаті пошуку користувач знайшовся – ми повертаємо помилку до фронтенд частини про те, що користувач з таким ідентифікатором вже існує. Переходимо на наступного кроку (див. рис. 3.12).

Як представлено на цьому рисунку, першим чином ми компонуємо масив запитів до AWS Rekognition, потім за допомогою функції Promise.allSettled, котра виконає всі запити зараз.

Promise.allSettled відрізняється від функції Promise.all тим, що в разі появи помилки в одному із промісів, всі інші проміси все одно виконаються і нам повернеться масив успішних і неуспішних запитів, на відміну від all – при помилці в тілі якого, всі інші проміси не виконаються, а Promise.all закінчиться помилкою першого невдалого промісу.

```
const faceIndexingPromisesArray: Promise<any>[] = userData.images.map(
    (img: string) => this.indexFaces(img, collection),
);
const responseOfFaceIndexing: PromiseSettledResult<IndexFacesCommandOutput>[] =
    await Promise.allSettled(faceIndexingPromisesArray);
const fullfilledFaceIndexingResponses: IndexFacesCommandOutput[] =
    responseOfFaceIndexing
        .filter(
            (item: PromiseSettledResult<IndexFacesCommandOutput>) =>
                item.status === 'fulfilled',
        )
        .map(
            (item: PromiseFulfilledResult<IndexFacesCommandOutput>) =>
                item.value,
        );
const faceIdArray: string[] = fullfilledFaceIndexingResponses
    .map((item: IndexFacesCommandOutput) => {
        return item.FaceRecords.map(
            (face: FaceRecord) => face.Face.FaceId,
        );
    })
    .flatMap((arr: string[]) => arr);
if (!faceIdArray?.length) {
    throw new HttpException(
        'Error: Face is not detected!',
        HttpStatus.BAD_REQUEST,
    );
}
```

Рисунок 3.12 – Код ідентифікації та валідації облич

Далі ми фільтруємо результати, які мають статус «fulfilled», а потім трансформуємо масив відповідей з сервісу AWS Rekognition у масив ідентифікованих FaceID, де перевіряємо чи масив має хоч один елемент. Якщо масив пустий, ми відправляємо помилку на фронтенд, сповіщаючи про те, що ні одне лице не було ідентифіковано сервісом AWS Rekognition. Переходимо до наступного кроку створення користувача (див. рис. 3.13).

```
const userExists: boolean = await this.searchUser(
    collection,
    newUser.username,
);
console.log('USER EXISTS: ', userExists);

if (!userExists) {
    await this.createAwsUser(collection, newUser.username);
}

const associatingResponse: AssociateFacesCommandOutput =
    await this.assignUserWithFaces(
        newUser.username,
        faceIdArray,
        collection,
    );
console.log('ASSOCIATING FACES...');
```

Рисунок 3.13 – Код створення UserID та асоціації облич

Впевнившись в тому, що сервіс штучного інтелекту все ж таки розпізнав хоча б одне обличчя, ми переходимо до створення UserID, сутності, яка виконує роль ідентифікації користувача і прив'язки зовнішнього сервісу (в нашому випадку AWS Rekognition) та її сутностей до внутрішньої екосистеми бекенд-додатку та бази даних. Спочатку ми перевіряємо чи не існує такого користувача в середовищі AWS Rekognition, якщо не існує – ми створюємо його, і після цього робимо запит на асоціацію отриманого UserID з масивом FaceIDs, що відіграє головну роль в авторизації користувача, так як ідентифіковані лиця будуть порівнюватись, і в разі співпадіння, ми повернемо UserID який був прив'язаний до ідентифікованих облич. Останнім кроком являється збереження користувача до бази даних та повернення результату виконання всієї функції (див. рис. 3.14).

```
294         const signedUpUser = await newUser.save();
295         return {
296             data: {
297                 faces: associatingResponseToReturn,
298                 user: signedUpUser,
299             },
300             status: HttpStatus.CREATED,
301         };
```

Рисунок 3.14 – Код створення користувача в базі даних і повернення результату виконання функції

Таким чином, був реалізований бекенд-додаток, який виконує функції збереження зображень, створення користувача, менеджменту користувачів та авторизації користувача в систему. Повний код цих модулів можна переглянути у Додатку В.

3.4 Огляд реалізованого додатку

3.4.1 Огляд функціоналу панелі адміністратора

Фінальним етапом реалізації додатку є його тестування, перевірка працездатності та переконання в тому, що він виконує поставлену мету роботи.

Першим чином перейдемо до сторінки створення користувача. Заповнимо всі поля необхідними даними, додамо персональне зображення користувача, а також завантажимо одне зображення по якому буде проходити ідентифікація обличчя користувача (див. рис. 3.15).

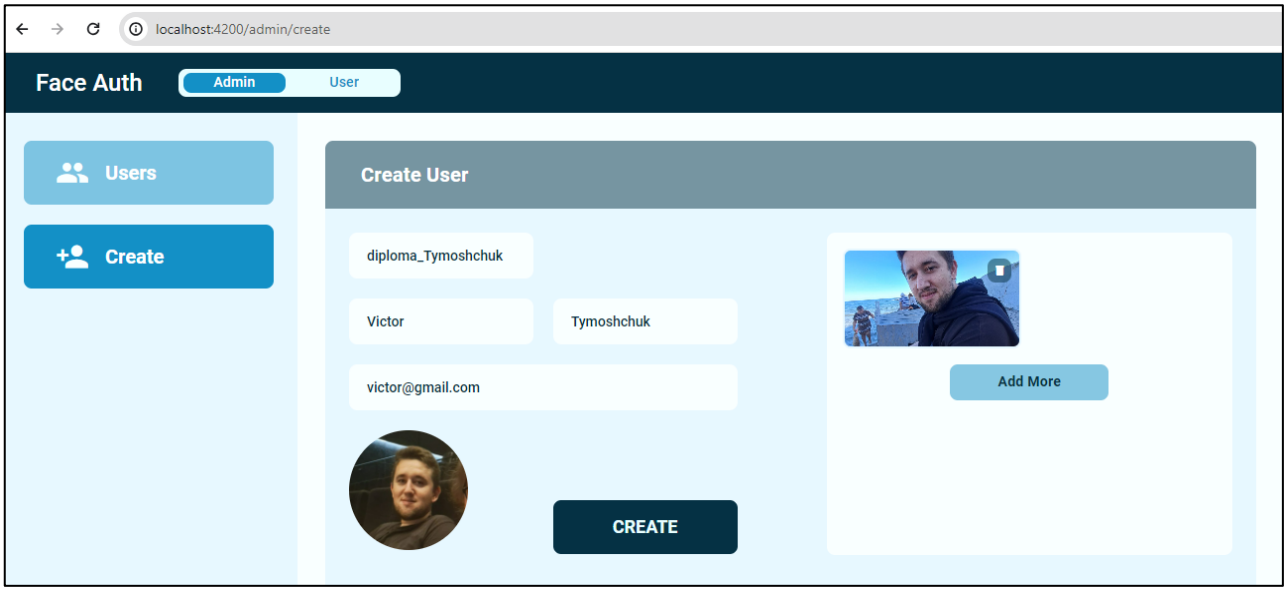

Рисунок 3.15 – Відображення заповнених полів вводу

Далі, ми натискаємо кнопку «CREATE», і після недовгого очікування, отримуємо сповіщення про успішне створення користувача. Клікаємо по кнопці «Users», що знаходиться в сайдбар-меню і переходимо до сторінки перегляду таблиці існуючих користувачів (див. рис. 3.16).

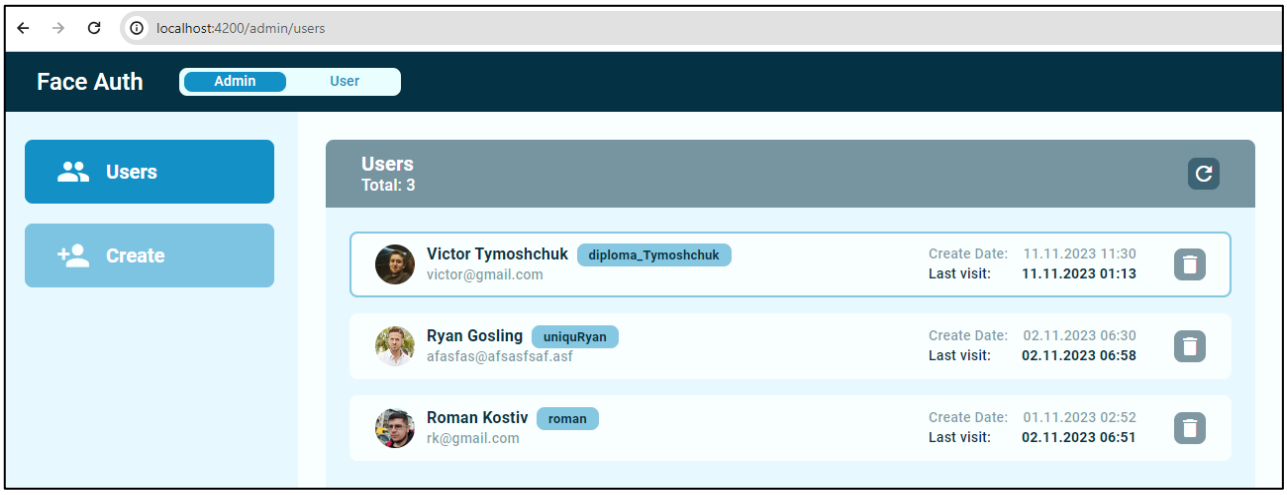

Рисунок 3.16 – Відображення сторінки таблиці користувачів

На даній сторінці ми відображаємо загальну кількість користувачів, список користувачів, персональні дані та дати створення і останнього візиту. Із функціональних можливостей: ми можемо видалити одного з користувачів і оновити таблицю, повторно запитавши список користувачів з бази даних, методом звернення до бекенд-додатку.

3.4.2 Огляд функціоналу сторінки авторизації

Перейдемо до сторінки авторизації користувача. Першим ділом нас зустріне вікно, в якому ми натискаємо кнопку «Sign In» та відразу переходимо до наступного кроку, де нам потрібно дозволити використовувати камеру девайсу та створити знімок обличчя (див. рис. 3.17).

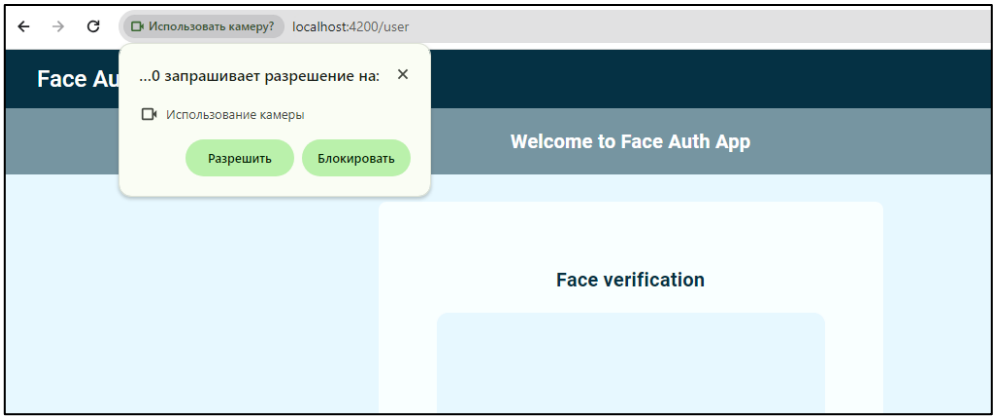

Рисунок 3.17 – Запит браузера на дозвіл використовувати камеру

Відразу після того, як ми дозволимо використовувати камеру, у відведеному вікні почнеться трансляція відео (див. рис. 3.18). Після цього, ми можемо спробувати захопити зображення з камери, щоб розпочати процес авторизації.

Також, цим функціоналом можна користуватись через мобільний телефон, додаток гарно працює на всіх платформах, коректно відображає трансляцію камери і підтримує адаптивну верстку.

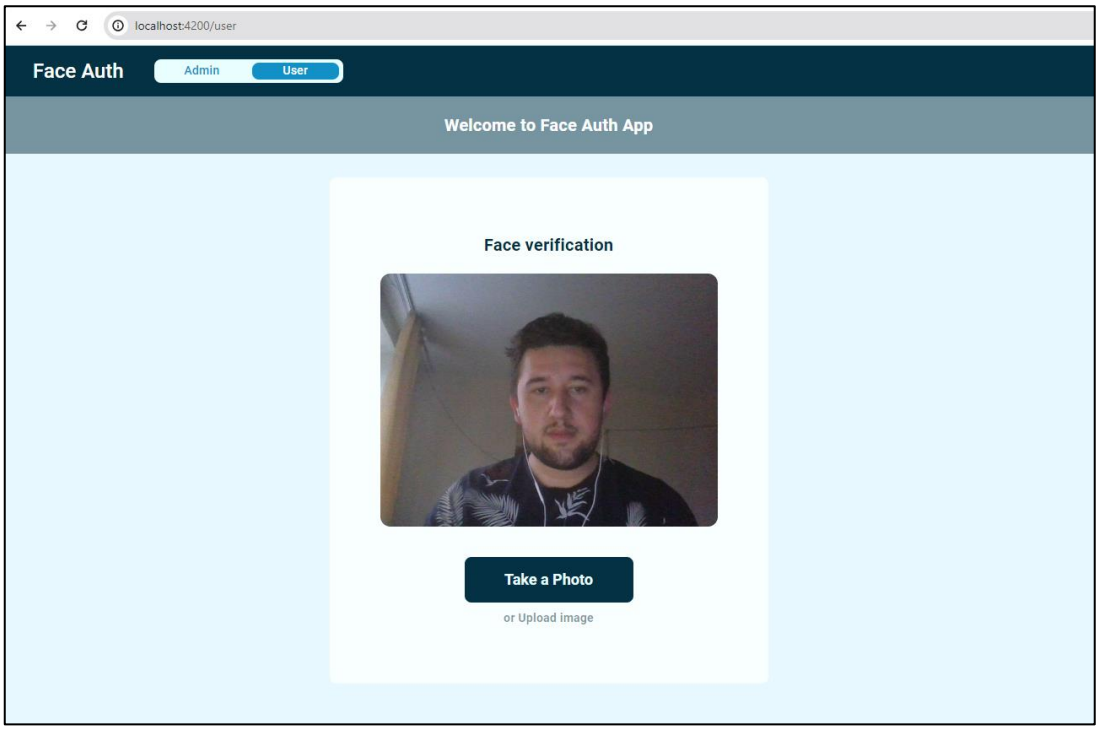

Рисунок 3.18 – Демонстрація трансляції відео з камери ноутбуку

Переглянути приклад використання додатку через телефон можна на рисунку 3.19.

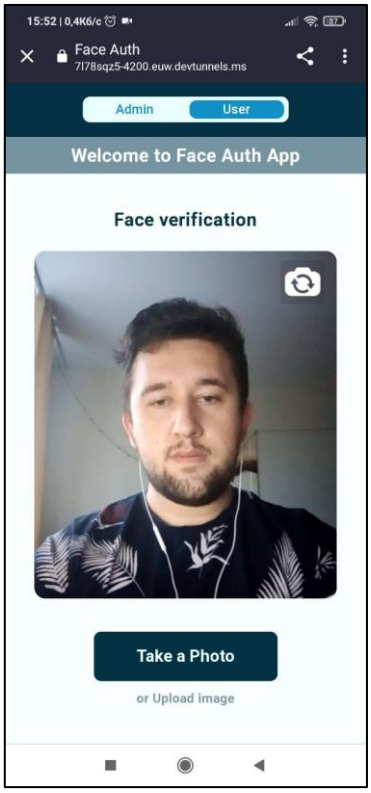

Рисунок 3.19 – Демонстрація трансляції відео з камери телефону

Після натискання на кнопку «Take a Photo», ми запустимо процес авторизації. Переглянемо відображення результату авторизації, отриманого з використанням ноутбуку (див. рис. 3.20).

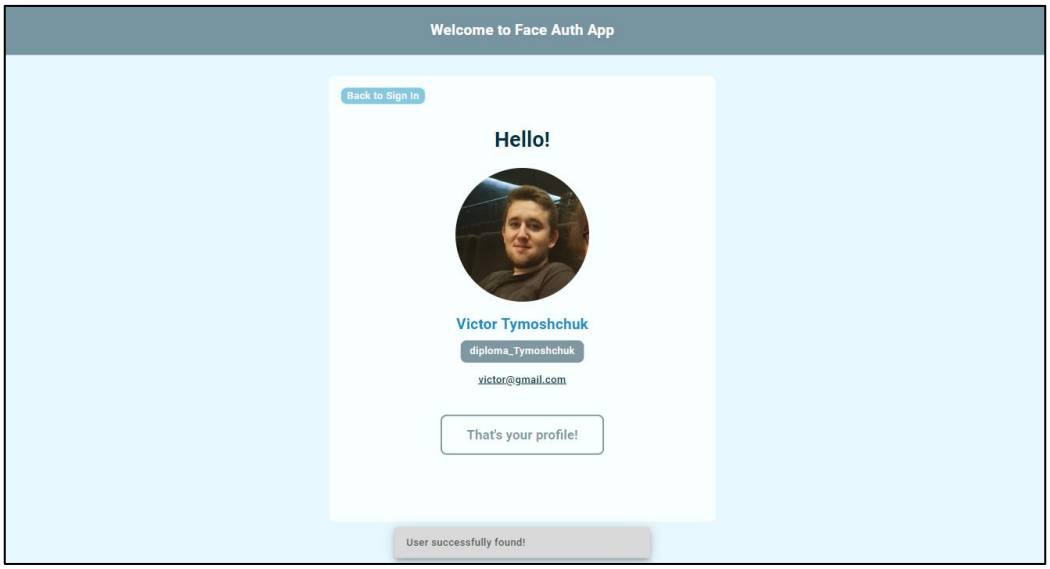

Рисунок 3.20 – Результат авторизації (через ноутбук)

Відображення результату авторизації через телефон можна переглянути на рисунку 3.21.

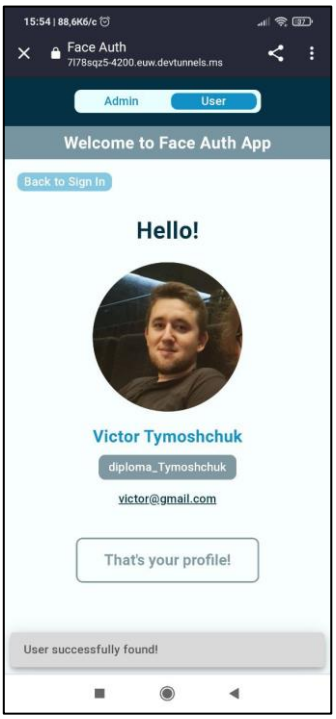

Рисунок 3.21 – Результат авторизації (через телефон)

Перевіривши працездатність реалізованої системи авторизації, виходячи з отриманих результатів, ми можемо сказати, що застосунок виконує поставлену перед ним задачу: від лиця адміністратора ми можемо створити обліковий запис користувача, а сам користувач може пройти ідентифікацію використовуючи лише власні біометричні дані, а саме – знімок свого обличчя, та авторизуватись в існуючу систему.

ВИСНОВКИ

В даній кваліфікаційній роботі було розроблено систему авторизації на основі розпізнавання обличчя з використанням штучного інтелекту AWS Rekognition. Дана робота була реалізована у вигляді кросбраузерного та кросплатформеного вебдодатку. Клієнтська частина системи була реалізована за допомогою фреймворку Angular, а серверна частина за допомогою NestJS.

Під час розробки додатку було проаналізовано більшість видів сучасних методів авторизації, їх переваги та недоліки. Слід відзначити, що досліджені методи мають доволі широке використання у всіх сферах нашого життя. Наприклад, блокчейн авторизація користується популярністю у криптовалютних або NFT-орієнтованих вебдодатках, так як дані вашого обліково запису знаходяться у децентралізованій базі даних, де вони не можуть бути якимось чином змінені, підроблені або викрадені. OAuth та OpenID використовуються зараз майже в кожному другому онлайн-сервісі, завдяки своїй зручності (можна проводити авторизацію в третій додаток, використовуючи, наприклад, свій Google-акаунт як постачальника даних) та доволі високого рівня безпеки (мається на увазі, що ви не надаєте ніяких логінів та паролів третій стороні). Що стосується методу авторизації на основі розпізнавання обличчя – його використання ми можемо побачити у технології FaceID від компаній Apple та Microsoft, але головною проблемою цього методу як раз і є відсутність широкого розповсюдження в екосистемі вебзастосунків. Під час дослідження даного типу авторизації, також зіткнувся з проблемою можливого обходу ідентифікації, методом підставки статичного зображення з обличчям до камери ідентифікуючого пристрою.

Виходячи з цього, у відповідності з метою роботи, була поставлена задача реалізувати таку систему авторизації, яку можна буде використовувати в будь-якому вебдодатку чи онлайн-сервісі, вирішивши проблеми недостатньої точності, а також обмеженого використання даного способу авторизації в різних

галузях нашого життя.

В процесі розробки даної системи авторизації було імплементовано функції створення користувачів, перегляду списку існуючих користувачів та їх видалення із системи (панель адміністратора), а також сам процес авторизації в систему користувачем.

Зокрема, було протестовано реалізований функціонал системи авторизації. На стороні бекенд-додатку було увімкнено логування, де ми могли переглядати результати процесу авторизації. Мали змогу спостерігати всі характеристики ідентифікованих облич, такі як схожість, розміщення частин обличчя відносно одна одної, контур, межі обличчя та інші. Сервіс AWS Rekognition добре описував усе, що притаманне зображенню та обличчю в цілому, використовуючи спеціальні теги. Таким чином, в реалізованому додатку ми змогли досягти високої точності в ідентифікації, а рефакторинг коду допоміг пришвидшити сам процес авторизації під час використання системи на практиці.

Як результат, авторизація в систему проходить дещо швидше у порівнянні з системами-аналогами, які використовують подібну технологію. У випадку, якщо системі підтасувати статичне зображення (спробувати обійти метод авторизації, наприклад, піднести фотографію обличчя яке зображено на паперовому знімку або ж з екрану мобільного телефону), то система ідентифікує такий варіант, опрацює його, та не допустить користувача до системи.

Отже, виходячи з вищезазначеного, результатом реалізації даної системи являється вебдодаток, який має високий потенціал до гнучкого масштабування функціоналу, є ефективним модулем авторизації в плані швидкості, безпечності і точності.

Виділивши серверну частину в окремий пакет та описавши документацію з використання, ми можемо поширити даний спосіб авторизації в мережі Інтернет для інших розробників, даючи можливість будь-якому підприємству чи організації реалізовувати ефективні додатки, які зможуть передувати за рахунок швидкої та безпечної авторизації, економлячи час клієнтів та підвищуючи якість користувацького досвіду даною системою.

ПЕРЕЛІК ПОСИЛАНЬ

1. Angular Developer Documentation. Introduction to the Angular docs. URL: https://angular.io/docs (дата звернення: 16.06.2023).

2. Rozentals N. Mastering TypeScript: Build enterprise-ready, modular web applications using TypeScript 4 and modern frameworks, 4th ed. Birmingham : Packt Publishing, 2021. 538 p.

3. NestJS Developer Documentation. URL: https://docs.nestjs.com (дата звернення: 24.07.2023).

4. Amazon Rekognition Developer Guide. Getting started with Amazon Rekognition. URL: https://docs.aws.amazon.com/rekognition/latest/dg/getting-started.html (дата звернення: 27.07.2023).

5. Figma Help Center. Introduction to design systems. URL: https://help.figma.com/hc/en-us (дата звернення: 27.07.2023).

6. RxJs. Reactive Extensions Library for JavaScript. Introduction. URL: https://rxjs.dev/guide/overview (дата звернення: 17.06.2023).

7. Angular Material. Getting Started with Angular Material. URL: https://material.angular.io (дата звернення: 17.06.2023).

8. Uluca D. Angular for Enterprise-Ready Web Applications: Build and deliver production-grade and cloud-scale evergreen web apps with Angular 9 and beyond, 2nd ed. Birmingham : Packt Publishing, 2020. 825 p.

9. Huber T. C. Getting Started with TypeScript: Includes Introduction to Angular. North Charlesto : CreateSpace Independent Publishing Platform, 2017. 178 p.

10. Murray N., Coury F., Lerner A., Taborda C. Ng-book, The Complete Guide to Angular. San Francisco : Fullstack.io, 2018. 685 p.

11. Wilken J. Angular in Action, Shelter Island. New York : Manning Publications Co., 2018. 320 p.

ДОДАТОК А

Код реалізації функціоналу завантаження зображень

Нижче наведено код сервісу завантаження зображень.

```
import { HttpErrorResponse } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable, catchError, map, of } from 'rxjs';
import { AppService } from 'src/app/app.service';

@Injectable({
  providedIn: 'root',
})
export class CreateUserService {
  constructor(private appService: AppService) {}

  componeRequestsToGetLinks(
    files: File[]
  ): Observable<{ url: string; fileName: string }>[] {
    const requestBody: Observable<{ url: string; fileName: string }>[] =
      files.map((file: File) =>
        this.appService.getLink(file).pipe(
          catchError((err: HttpErrorResponse) => {
            return of({ url: '', fileName: '' });
          }),
          map((response: { url: string }) => {
            return {
              url: response.url,
              fileName: file.name,
            };
          })
        )
      );
```

```
  return requestBody;
}

componeLinksToPutFiles(
  urls: { url: string; fileName: string }[],
  files: File[]
): Observable<{ imageUrl: string; completed: boolean }>[] {
  console.log(urls);

  const pairs: { url: string; file: File | undefined }[] = urls.map(
    (item: { url: string; fileName: string }) => {
      return {
        file: files.find((file: File) => file.name == item.fileName),
        url: item.url,
      };
    }
  );
  const requestBody: Observable<{ imageUrl: string; completed: boolean }>[] =
    pairs.map(
      (
        item: { url: string; file: File | undefined },
        i: number,
        arr: { url: string; file: File | undefined }[]
      ) =>
        this.appService.uploadImage(item.url, item.file).pipe(
          catchError((err: HttpErrorResponse) => {
            return of();
          }),
          map<any, { imageUrl: string; completed: boolean }>((res: any) => {
            return {
              imageUrl: item.url,
              completed: arr.length - 1 === i ? true : false,
            };
          })
        )
```

```
      );
    return requestBody;
  }
}
```

ДОДАТОК Б

Код реалізації фронтенд частини

Б.1 Код компоненту create-user.component.ts

```
import { Component } from '@angular/core';
import { Observable, catchError, concat, forkJoin, of } from 'rxjs';
import { HttpErrorResponse } from '@angular/common/http';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { MatSnackBar } from '@angular/material/snack-bar';
import { AppService } from 'src/app/app.service';
import { SnackbarComponent } from '../admin.component';
import { CreateUserService } from './create-user.service';

@Component({
  selector: 'create-user',
  templateUrl: './create-user.component.html',
  styleUrls: ['./create-user.component.scss']
})
export class CreateUserComponent {
  loading: boolean = false;
  imgLoading: boolean = false;
  imgToSave: string = '';
  img: string = '';
  gallery: string[] = [];
  imgToRekognize: string[] = [];

  fg = new FormGroup({
    username: new FormControl('', Validators.required),
    firstName: new FormControl('', Validators.required),
    lastName: new FormControl('', Validators.required),
    email: new FormControl('', Validators.required),
  })
```

```
    constructor(private appService: AppService, private _snackBar: MatSnackBar, private createUserService: CreateUserService) {
      this.appService.getUser().subscribe();
    }

    onFileSelected(files: FileList) {
      const filesArr: File[] = Array.from(files);
      this.imgLoading = true;
      const requests: Observable<{url: string, fileName: string}>[] = this.createUserService.componeRequestsToGetLinks(filesArr);
      forkJoin<{url: string, fileName: string}[]>({...requests}).pipe(
        catchError((err: HttpErrorResponse) => {
          this.imgLoading = false;
          return of();
        })
      ).subscribe((data: {url: string, fileName: string}[]) => {
        console.log(data);
        const pairedFilesAndUrls: {url: string, fileName: string}[] = Object.values(data);
        console.log(pairedFilesAndUrls);
        this.uploadImages(pairedFilesAndUrls, filesArr);
      });
    }

    uploadImages(urls: {url: string, fileName: string}[], files: File[]) {
      const requests: Observable<{imageUrl: string, completed: boolean}>[] = this.createUserService.componeLinksToPutFiles(urls, files);
      concat<{imageUrl: string, completed: boolean}[]>(...requests).pipe(
        catchError((err: HttpErrorResponse) => {
          this.imgLoading = false;
          return of();
        })
      ).subscribe((response: {imageUrl: string, completed: boolean}) => {
        console.log(response);
        let formattedUrl: URL = new URL(response.imageUrl);
        let imgForRequest: string = formattedUrl.pathname.replace('/', '');
```

```
    let imgToDisplay: string = formattedUrl.origin + formattedUrl.pathname;
    this.gallery.push(imgToDisplay);
    this.imgToRekognize.push(imgForRequest);
    if (response.completed) {
      this.imgLoading = false;
    }
  })
}

removeImage(index: number) {
  this.imgToRekognize.splice(index, 1);
  this.gallery.splice(index, 1);
}


onPhotoSelected(event: any) {
  console.log(event.target.files);
  this.loading = true;
  const file: File = event.target.files[0];
  this.appService.getLink(file).pipe(
    catchError((err: HttpErrorResponse) => {
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: {url: string}) => {
    if (!data) return;
    this.uploadImg(data.url, file);
  })
}

uploadImg(url: string, file: File) {
  console.log(url);
  const formattedUrl: URL = new URL(url);
  this.appService.uploadImage(url, file).pipe(
```

```
    catchError((err: HttpErrorResponse) => {
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: any) => {
    this.img = formattedUrl.origin + formattedUrl.pathname;
    this.loading = false;
  });
}

createUser() {
  if (this.fg.invalid) return;
  if (!this.imgToRekognize.length) return;
  this.loading = true;
  const body: any = this.fg.value;
  body.images = this.imgToRekognize;
  body.userPhoto = this.img || null;
  this.appService.createUser(body).pipe(
    catchError((err: HttpErrorResponse) => {
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: any) => {
    console.log(data);
    this.openSnackBar('Successfully created!');
    this.loading = false;
  })
}

openSnackBar(text: string) {
  this._snackBar.openFromComponent(SnackbarComponent, {
    duration: 6000,
    data: text
```

```
    });
  }
}
```

Б.2 Код компоненту user-list.component.ts

```
import { Component } from '@angular/core';
import { MatSnackBar } from '@angular/material/snack-bar';
import { AppService } from 'src/app/app.service';
import { FAUser } from 'src/app/interfaces/user.interface';
import { SnackbarComponent } from '../admin.component';

@Component({
  selector: 'users-list',
  templateUrl: './users-list.component.html',
  styleUrls: ['./users-list.component.scss']
})
export class UsersListComponent {
  loading: boolean = false;
  total: number = 0;
  users: any[] = [];

  constructor(private appService: AppService, private _snackBar: MatSnackBar) {
    this.getUsers();
  }

  getShortName(user: FAUser) {
    return user?.firstName?.slice(0, 1) + user?.lastName?.slice(0, 1) || '';
  }

  getUsers() {
    this.loading = true;
    this.appService.getUser().subscribe((data: any) => {
```

```
      console.log(data);
      this.users = data.data.map((user: FAUser) => {
        return {
          ...user,
          loading: false
        }
      });
      this.total = data.total;
      this.loading = false;
    });
  }

  deleteUser(user: any) {
    user.loading = true;
    const body: {username: string} = { username: user.username };
    this.appService.deleteUser(body).subscribe((data: any) => {
      console.log(data);
      if (data) {
        user.loading = false;
        const index = this.users.findIndex((item) => item.username == user.username);
        this.users.splice(index, 1);
        this.openSnackBar("You've successfully deleted the user");
      }
    })
  }

  openSnackBar(text: string) {
    this._snackBar.openFromComponent(SnackbarComponent, {
      duration: 6000,
      data: text
    });
  }

}
```

Б.3 Код компоненту user-sign-in.component.ts

```
import { Component } from '@angular/core';
import { AppService } from '../app.service';
import { Observable, Subject, catchError, of } from 'rxjs';
import { HttpErrorResponse } from '@angular/common/http';
import { MatSnackBar } from '@angular/material/snack-bar';
import { SnackbarComponent } from '../admin/admin.component';
import { WebcamImage } from 'ngx-webcam';
import { FAUser } from '../interfaces/user.interface';

@Component({
  selector: 'user-sign-in',
  templateUrl: './user-sign-in.component.html',
  styleUrls: ['./user-sign-in.component.scss']
})
export class UserSignInComponent {
  loading: boolean = false;
  imgToSave: string = '';
  img: string = '';
  user: FAUser | undefined = undefined;
  errorText: string = '';
  signInStep: 'welcome' | 'photo' | 'result' = 'welcome';
  private trigger: Subject<any> = new Subject();
  public webcamImage!: WebcamImage;
  private nextWebcam: Subject<any> = new Subject();
  sysImage = '';

  constructor(private appService: AppService, private _snackBar: MatSnackBar) {}

  onFileSelected(event: any) {
    this.errorText = '';
    const fileReader = new FileReader();
    fileReader.onload = (evt: any) => {
      let base64: string = evt.target.result;
```

```
    this.img = base64;
  }
  fileReader.readAsDataURL(event[0]);
  let file: File = event[0];
  this.loading = true;
  this.appService.getLink(file).pipe(
    catchError((err: HttpErrorResponse) => {
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: {url: string}) => {
    if (!data) return;
    this.uploadImg(data.url, file);
  })
}
uploadImg(url: string, file: File) {
  console.log(url);
  const formattedUrl: URL = new URL(url);
  this.imgToSave = formattedUrl.pathname;

  this.appService.uploadImage(url, file).pipe(
    catchError((err: HttpErrorResponse) => {
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: any) => {
    this.loading = false;
    this.authenticateUser();
  });
}
authenticateUser() {
  this.errorText = '';
  console.log('CREATE');
```

```
  if (!this.imgToSave) return;
  this.loading = true;
  const body: {img: string} = { img: this.imgToSave.replace('/', '') };
  this.appService.authenticateUser(body).pipe(
    catchError((err: HttpErrorResponse) => {
      this.backToPhotoStep();
      this.openSnackBar(err.error.message || err.message);
      this.loading = false;
      return of();
    })
  ).subscribe((data: FAUser) => {
    console.log(data);
    this.openSnackBar('User successfully found!');
    this.user = data;
    this.signInStep = 'result';
    this.loading = false;
  })
}
backToPhotoStep() {
  this.user = undefined;
  this.sysImage = '';
  this.img = '';
  this.imgToSave = '';
  this.signInStep = 'photo';
}
public getSnapshot(): void {
  this.trigger.next(void 0);
}
public captureImg(webcamImage: WebcamImage): void {
  this.webcamImage = webcamImage;
  this.sysImage = webcamImage!.imageAsDataUrl;
  this.createFileAndSignIn();
}
public get invokeObservable(): Observable<any> {
  return this.trigger.asObservable();
```

```
  }
  public get nextWebcamObservable(): Observable<any> {
    return this.nextWebcam.asObservable();
  }
  generateRandomText(length: number): string {
    const possibleChars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
    let result = '';
    for (let i = 0; i < length; i++) {
      result += possibleChars.charAt(Math.floor(Math.random() * possibleChars.length));
    }
    return result;
  }
  createFileAndSignIn() {
    const filename: string = this.generateRandomText(10) + '.png';
    const file = new File([this.convertDataUrlToBlob(this.sysImage)], filename, {type: `image/png`});
    this.onPhotoTaked(file);
  }
  onPhotoTaked(file: File) {
    this.loading = true;
    this.appService.getLink(file).pipe(
      catchError((err: HttpErrorResponse) => {
        this.openSnackBar(err.error.message || err.message);
        this.loading = false;
        return of();
      })
    ).subscribe((data: {url: string}) => {
      if (!data) return;
      this.uploadImg(data.url, file);
    })
  }
  convertDataUrlToBlob(dataUrl: string): Blob {
    const arr = dataUrl.split(',');
    const mime = arr[0].match!(/:(.*?);/)![1];
```

```
    const bstr = atob(arr[1]);
    let n = bstr.length;
    const u8arr = new Uint8Array(n);
    while (n--) {
        u8arr[n] = bstr.charCodeAt(n);
    }
    return new Blob([u8arr], {type: mime});
  }
  openSnackBar(text: string) {
    this._snackBar.openFromComponent(SnackbarComponent, {
      duration: 7000,
      data: text
    });
  }
  getShortName(user: FAUser | undefined) {
    console.log(user);
    if (!user) return '';
    return user?.firstName?.slice(0, 1) + user?.lastName?.slice(0, 1) || '';
  }
}
```

ДОДАТОК В

Код реалізації бекенд частини

Нижче наведено код створення користувача, отримання списку користувачів та авторизації користувача.

```
import {
  HttpException,
  HttpStatus,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import {
  RekognitionClient,
  CreateCollectionCommandInput,
  CreateCollectionCommand,
  ListCollectionsCommandInput,
  ListCollectionsCommand,
  CreateUserCommandInput,
  CreateUserCommand,
  IndexFacesCommandInput,
  IndexFacesCommand,
  ListFacesCommandInput,
  ListFacesCommand,
  IndexFacesCommandOutput,
  Face,
  FaceRecord,
  AssociateFacesCommandInput,
  AssociateFacesCommand,
  AssociateFacesCommandOutput,
  SearchUsersByImageCommandInput,
  SearchUsersByImageCommand,
```

```
    SearchUsersByImageCommandOutput,
    UserMatch,
    UnsuccessfulFaceAssociation,
    AssociatedFace,
    ListFacesCommandOutput,
    SearchUsersCommandInput,
    SearchUsersCommand,
    DeleteUserCommandInput,
    DeleteUserCommand,
    DeleteFacesCommandOutput,
    DeleteFacesCommandInput,
    DeleteFacesCommand,
} from '@aws-sdk/client-rekognition';
import { InjectModel } from '@nestjs/mongoose';
import { User, UserDocument } from './schemas/user.schema';
import { Model } from 'mongoose';

@Injectable()
export class AwsService {
    private client = new RekognitionClient({
        region: this.configService.getOrThrow('AWS_S3_REGION'),
    });
    private readonly bucket: string =
        this.configService.getOrThrow('AWS_BUCKET_NAME');
    private readonly collection: string = this.configService.getOrThrow(
        'AWS_COLLECTION_NAME',
    );

    constructor(
        private readonly configService: ConfigService,
        @InjectModel(User.name) private userModel: Model<UserDocument>,
    ) {}

    async createCollectionAndRetrive(): Promise<string> {
        const existingCollections: any = await this.retriveCollections();
```

```
    const collectionName: string = this.collection;
    if (existingCollections.CollectionIds.includes(collectionName)) {
      console.log('EXISTS');
      return collectionName;
    }
    const input: CreateCollectionCommandInput = {
      CollectionId: collectionName,
      Tags: {
        auth: 'face',
      },
    };
    const command = new CreateCollectionCommand(input);
    let response: string = collectionName;
    try {
      await this.client.send(command);
    } catch (err) {
      throw new HttpException('Forbidden', HttpStatus.BAD_REQUEST);
    }
    console.log('response: ', response);
    return response;
  }

  async retriveCollections() {
    const input: ListCollectionsCommandInput = {
      MaxResults: 10,
    };
    const command = new ListCollectionsCommand(input);
    const response = await this.client.send(command);
    return response;
  }

  async listFaces(
    id: string,
    collection: string,
  ): Promise<ListFacesCommandOutput> {
```

```
    const input: ListFacesCommandInput = {
      CollectionId: collection,
      UserId: id,
    };
    const command = new ListFacesCommand(input);
    return await this.client.send(command);
  }

  async deleteFaces(
    collection: string,
    faces: string[],
  ): Promise<DeleteFacesCommandOutput> {
    const input: DeleteFacesCommandInput = {
      CollectionId: collection,
      FaceIds: faces,
    };
    const command = new DeleteFacesCommand(input);
    return await this.client.send(command);
  }

  async indexFaces(img: string, collection: string): Promise<any> {
    const input: IndexFacesCommandInput = {
      CollectionId: collection,
      Image: {
        S3Object: {
          Bucket: this.bucket,
          Name: img,
        },
      },
    };
    const command = new IndexFacesCommand(input);
    return await this.client.send(command);
  }

  async searchUser(collection: string, userId: string): Promise<boolean> {
```

```
    const input: SearchUsersCommandInput = {
        CollectionId: collection,
        UserId: userId,
    };
    const command = new SearchUsersCommand(input);
    let response: boolean = false;
    try {
        await this.client.send(command);
        response = true;
    } catch (err) {
        response = false;
    }
    return response;
}

async updateLastVisit(data: { username: string; }) {
    const user = await this.userModel.findOne({
        username: data.username,
    });
    if (!user) {
        throw new UnauthorizedException('User not found');
    }
    return this.userModel.updateOne({username: data.username}, { lastVisit: Date.now() });
}

async createUser(userData: {
    username: string;
    firstName: string;
    lastName: string;
    userPhoto: string | null;
    email: string;
    images: string[];
}) {
    const user = await this.userModel.findOne({
        username: userData.username,
```

```
        });
        if (user) {
            throw new UnauthorizedException('User already exists');
        }
        const newUser = new this.userModel({
            ...userData,
            image: undefined,
        });
        const collection: string = await this.createCollectionAndRetrive();
        const faceIndexingPromisesArray: Promise<any>[] = userData.images.map(
            (img: string) => this.indexFaces(img, collection),
        );
        const  responseOfFaceIndexing:  PromiseSettledResult<IndexFacesCommandOutput>[]
=
            await Promise.allSettled(faceIndexingPromisesArray);
        const fullfilledFaceIndexingResponses: IndexFacesCommandOutput[] =
            responseOfFaceIndexing
                .filter(
                    (item: PromiseSettledResult<IndexFacesCommandOutput>) =>
                        item.status === 'fulfilled',
                )
                .map(
                    (item: PromiseFulfilledResult<IndexFacesCommandOutput>) =>
                        item.value,
                );
        const faceIdArray: string[] = fullfilledFaceIndexingResponses
            .map((item: IndexFacesCommandOutput) => {
                return item.FaceRecords.map(
                    (face: FaceRecord) => face.Face.FaceId,
                );
            })
            .flatMap((arr: string[]) => arr);
        if (!faceIdArray?.length) {
            throw new HttpException(
                'Error: Face is not detected!',
```

```
        HttpStatus.BAD_REQUEST,
    );
}
const userExists: boolean = await this.searchUser(
    collection,
    newUser.username,
);
console.log('USER EXISTS: ', userExists);
if (!userExists) {
    await this.createAwsUser(collection, newUser.username);
}
const associatingResponse: AssociateFacesCommandOutput =
    await this.assignUserWithFaces(
        newUser.username,
        faceIdArray,
        collection,
    );
console.log('ASSOCIATING FACES...');
const associatingResponseToReturn: {
    status:
        | 'fully-associated'
        | 'partially-associated'
        | 'not-associated'
        | 'empty';
    usersAlreadyAssociated: string[];
} = this.checkAssociations(associatingResponse);
if (
    associatingResponseToReturn.status === 'not-associated' ||
    associatingResponseToReturn.status === 'empty'
) {
    return {
        data: {
            faces: associatingResponseToReturn,
            user: null,
        },
```

```
            status: HttpStatus.FORBIDDEN,
        };
    }
    const signedUpUser = await newUser.save();
    return {
        data: {
            faces: associatingResponseToReturn,
            user: signedUpUser,
        },
        status: HttpStatus.CREATED,
    };
}

checkAssociations(associatingResponse: AssociateFacesCommandOutput): {
    status:
        | 'fully-associated'
        | 'partially-associated'
        | 'not-associated'
        | 'empty';
    usersAlreadyAssociated: string[];
} {
    const usersWithUnsuccessfullyAssociatedFaces: string[] =
        associatingResponse.UnsuccessfulFaceAssociations.map(
            (item: UnsuccessfulFaceAssociation) => item.UserId,
        );
    const successfullyAssociatedFaces: string[] =
        associatingResponse.AssociatedFaces.map(
            (item: AssociatedFace) => item.FaceId,
        );
    let status:
        | 'fully-associated'
        | 'partially-associated'
        | 'not-associated'
        | 'empty' = 'empty';
    let usersAlreadyAssociated: string[] = [];
```

```
    if (successfullyAssociatedFaces.length) {
        status = 'fully-associated';
    }
    if (usersWithUnsuccessfullyAssociatedFaces.length) {
        usersAlreadyAssociated = usersWithUnsuccessfullyAssociatedFaces;
        if (successfullyAssociatedFaces.length) {
            status = 'partially-associated';
        } else {
            status = 'not-associated';
        }
    }
    return {
        status: status,
        usersAlreadyAssociated: usersAlreadyAssociated,
    };
}

async createAwsUser(collection: string, username: string) {
    const input: CreateUserCommandInput = {
        CollectionId: collection,
        UserId: username,
    };
    const command = new CreateUserCommand(input);
    const response = await this.client.send(command);
    return response;
}

async deleteAwsUser(collection: string, username: string) {
    const input: DeleteUserCommandInput = {
        CollectionId: collection,
        UserId: username,
    };
    const command = new DeleteUserCommand(input);
    const response = await this.client.send(command);
    return response;
```

```
    }

    async assignUserWithFaces(
      userId: string,
      faces: string[],
      collection: string,
    ) {
      const input: AssociateFacesCommandInput = {
        CollectionId: collection,
        FaceIds: faces,
        UserId: userId,
      };
      const command = new AssociateFacesCommand(input);
      const response = await this.client.send(command);
      return response;
    }

    async signIn(img: string) {
      const input: SearchUsersByImageCommandInput = {
        CollectionId: this.collection,
        Image: {
          S3Object: {
            Bucket: this.bucket,
            Name: img,
          },
        },
      };
      const command = new SearchUsersByImageCommand(input);
      const response: SearchUsersByImageCommandOutput = await
this.client.send(command);
      let result: string = '';
      const matchedUser: UserMatch | undefined = response.UserMatches.find(
        (user: UserMatch) => {
          console.log('USER SIMILARITY: ', user.Similarity);
          if (user.Similarity > 96) {
```

```
            return user;
          } else {
            return undefined;
          }
        },
    );
    if (!matchedUser) {
      result = 'User that mathches the uploaded image is not found.';
      throw new HttpException(result, HttpStatus.BAD_REQUEST);
    }
    return await this.getUser(matchedUser.User.UserId);
}

async getUser(id: string) {
    const user = await this.userModel.findOne({ username: id });
    if (!user) {
      throw new UnauthorizedException('User does not exist');
    }
    await this.updateLastVisit({username: id});
    return user;
}
async getUsers() {
    const users = await this.userModel.find().sort({ createdAt: 'desc' });
    const total = await this.userModel.countDocuments();
    return {
      data: users,
      total: total,
    };
}

async deleteUser(body: { username: string }) {
    const user = await this.userModel.findOne({ username: body.username });
    if (!user) {
      throw new UnauthorizedException('User does not exist');
    }
```

```
        const collection: string = await this.createCollectionAndRetrive();
        const userExists: boolean = await this.searchUser(
          collection,
          body.username,
        );
        if (!userExists) {
          await this.userModel.findOneAndDelete({ username: body.username });
          throw new HttpException(
            'AWS user not found',
            HttpStatus.BAD_REQUEST,
          );
        }
        const usersFaces: ListFacesCommandOutput = await this.listFaces(
          body.username,
          collection,
        );
        const faceIdsToDelete: string[] = usersFaces.Faces.map(
          (face: Face) => face.FaceId,
        );
        if (faceIdsToDelete.length) {
          await this.deleteFaces(collection, faceIdsToDelete);
        }
        await this.deleteAwsUser(collection, body.username);
        return await this.userModel.findOneAndDelete({
          username: body.username,
        });
      }

}
```