МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ «АВТОМОБІЛЬНА ПАРКОВКА»**

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)
О.Ю. Суханов
(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

Суханову Олександру Юрійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інформаційної системи «Автомобільна парковка»

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 інформаційної системи «Автомобільна парковка».

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.	17.05.2023	
2.	Збір вихідних даних.	07.06.2023	
3.	Обробка методичних та теоретичних	28.06.2023	
	джерел.		
4.	Розробка першого розділу.	30.08.2023	
5.	Розробка другого розділу та третього розділу.	08.11.2023	
6.	Оформлення та нормоконтроль	20.11.2023	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	15.12.2023	

Студент _____ О.Ю. Суханов
(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 інформаційної системи «Автомобільна парковка»: 61 с., 34 рис., 10 джерел, 4 додатки.

ВЕРИФІКАЦІЯ, ANGULAR, AWS REKOGNITION, NESTJS, TYPESCRIPT, RXJS, UML.

Об’єкт дослідження – процес верифікації з допомогою сервісів розпізнавання тексту.

Мета роботи: розробити інформаційної систему «Автомобільна парковка».

Методи дослідження – моделювальний, аналітичний, програмний.

У сфері автомобільного транспорту верифікація використовується для контролю доступу до паркувальних місць, оплати паркування та інших цілей. На сьогоднішній день існує ряд технологій, які дозволяють автоматизувати процес верифікації.

Таким чином, за результатами роботи створено просту та ефективну інформаційну систему паркування з використанням Angular, NEST.JS та AWS Rekognition.

SUMMARY

Master’s qualifying paper «Development of the Car Parking Infomaton System»: 61 pages, 34 figures, 10 references, 4 supplements.

VERIFICATION, ANGULAR, AWS REKOGNITION, NESTJS, TYPESCRIPT, RXJS, UML.

The object of the study is the verification process using text recognition services.

The aim of the study is to develop an information system “Car Parking”.

The methods of research are modeling, analytical reasoning, and programming approaches.

In the field of automotive transportation, verification is used for controlling access to parking spaces, payment for parking, and other purposes. Currently, there are several technologies available that allow automating the verification process. Consequently, the work resulted in creating a simple and efficient parking information system using Angular, NEST.JS, and AWS Rekognition.

ЗМІСТ

ВСТУП

Сучасність змінюється та розвивається швидкими темпами і це охоплює майже усі галузі від науки, технологій та бізнесу до повсякденного життя людей. В великих містах ці зміни помітно найбільше, тому що там проживає багато людей і через це інфраструктура міста постійно потребує розвитку та впровадження інноваційних рішень. Однією з постійних проблем міст є паркування автомобілів.

Постійне зростання кількості автомобілів у містах призводить до того, що паркувальні місця переповнюються, а знайти вільне місце стає все складніше. Це викликає незручності для водіїв, через що втрачається багато часу, нервів та енергії. Також через складність знайти вільне місце, часто доводиться паркуватися в не відведеному для паркування місці, через що можна отримати штраф або навіть завдати шкоду оточуючим.

Роблячи висновок з вище написаного було прийнято рішення про створення спеціальної інформаційної системи автомобільної парковки, яка б покращила автотранспортну логістику та була простою і зрозумілою у використанні.

Автоматизована система паркування полегшить і пришвидшить процес паркування для водіїв, сприятиме зменшенню транспортних заторів, забезпечить ефективне використання паркувальних місць та зменшить ризик присвоєння чужого паркувального місця. Забезпечення легкості та зручності у паркуванні допоможе покращити міську мобільність та впорядкувати транспортну інфраструктуру. Це не лише технологічний крок у майбутнє транспортної логістики, а й важливий крок у створенні комфортної та модернізованої міської інфраструктури.

Метою дипломної роботи є розробка інформаційної системи, яка буде інформувати про наявність автомобіля (користувача) в певній базі користувачів та пропускати або ні в залежності від статусу користувача (зареєстрований/не

зареєстрований). Така система допоможе покращити та модернізувати процес паркування.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– провести аналіз предметної області майбутньої інформаційної системи;

– дослідити та порівняти існуючі системи розпізнавання тексту;

– обрати потрібні технології та інструменти для реалізації проєкту;

– створити діаграми, які описують майбутню систему;

– спроєктувати макет застосунку.

Об’єкт дослідження – процес верифікації з допомогою сервісів розпізнавання тексту.

Методи дослідження: моделювальний, аналітичний, програмний.

У наступних розділах ми детально розглянемо структуру та процес розробки майбутньої інформаційної системи з використанням таких технології як Angular, NestJS та сервісу для розпізнавання тексту.

Така система автомобільної парковки унікальна через свій підхід до вирішення актуальної проблеми міського паркування. Вона використовує передові технології розпізнавання тексту та інформаційні системи для ефективного керування доступом до паркувальних місць.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ІНФОРМАЦІЙНОЇ СИСТЕМИ

1.1 Опис предметної області

Предметною областю є розробка інформаційної системи, яка буде надавати послуги з паркування автомобілів. Система повинна верифікувати користувачів і в залежності від статусу користувача в системі, пропускати його на територію парковки або ні. Взаємодія відбувається наступним чином: користувача попередньо реєструють в системі, потім за ним і його автомобілем закріплюється відповідне паркувальне місце.

Далі користувачу потрібно пройти верифікацію в системі. Це відбувається за таким сценарієм. Спочатку робиться знімок, потім на цьому зображенні за допомогою технологій розпізнавання тексту знаходиться номер машини і передається в систему для верифікації користувача, якщо користувач є в системі доступ до парковки дозволяється, в іншому випадку доступ забороняється.

1.2 Дослідження технології розпізнавання тексту

Одним з головних запитань під час створення інформаційної системи для верифікації за номером автомобіля є те, а як саме ми маємо отримувати номер машини для її верифікації в системі. Основною технологією для розпізнавання тексту є оптичне розпізнавання символів.

Оптичне розпізнавання символів – це переведення зображень рукописного, друкованого та машинописного тексту в послідовність кодів, які використовуються для представлення в текстовому редакторі. Переведення відбувається механічним або електронним способом. Розпізнавання тексту пришвидшує конвертацію книг та документів в електронний вигляд, покращує автоматизацію систем обліку в бізнесі, дозволяє редагувати текст, аналізувати

інформацію та навіть допомагає застосовувати до тексту електронний переклад і багато чого іншого. Існує декілька технологій, які використовуються в ORC. Розглянемо основні з них.

Шаблонний пошук. Це метод порівняння символів зображення з попередньо визначеними шаблонами відомих символів. Він добре працює для друкованого тексту з обмеженими варіаціями шрифту та розмір.

Нейронні мережі. Це обчислювальні системи з потужними алгоритмами машинного навчання, які можуть навчитися розпізнавати символи, аналізуючи великий набір даних зображень із відповідним текстом. Цей метод є більш точним, ніж метод шаблонного пошуку, і може обробляти ширший діапазон шрифтів і стилів, включаючи рукописний текст.

Приховані марковські моделі. Це статистичні моделі, які використовуються для аналізу послідовностей подій. У ORC приховані марковські моделі можуть використовуватися для аналізу послідовності пікселів на зображенні для ідентифікації символів.

Провівши аналіз описаних технології які використовуються в ORC, можна прийти до висновку, що найкращим варіантом для розпізнавання тексту є нейромережі.

Нейромережі використовують для вирішення великої кількості різноманітних завдань, від розпізнавання об’єктів до прогнозування якихось подій та багато іншого. Проте нейромережа це складна система, яку важко імплементувати самотужки і це займе надто багато часу та фінансів. До того ж потрібно мати дуже великий об’єм навчальних даних, щоб натренувати модель для коректної роботи і також це потребує чималих обчислювальних ресурсів, а такий варіант нам не підходить, наш пріоритет це легка система, яка може працювати практично в будь-яких умовах.

Тому було прийнято рішення про використання однієї з вже існуючи систем розпізнавання тексту та подальшого використання в нашому проєкті. Це пришвидшить нашу роботу та зекономить нам купу ресурсів. Але для початку проведемо порівняння цих систем та визначимо найкращу систему під наші задачі.

1.3 Порівняння існуючих систем розпізнавання тексту

Майже всі технології описані в цьому підрозділі мають безліч функцій для розпізнавання тексту, об’єктів та навіть обличчя, і також в своєму арсеналі вони мають головну для нас функцію, це розпізнавання номер знаків.

Для того, щоб обрати найкращу з представлених технології розпізнавання нам потрібно зробити порівняння кожної з них. Але спочатку напишемо характеристику до кожної з них і в кінці зробимо висновки.

1.3.1 Platerecognizer

Platerecognizer – це open-source LPR-платформа, яка дозволяє кожному створювати свої системи розпізнавання. Платформа пропонує широкий спектр функцій, включаючи:

– підтримку різних типів камер;
– підтримку різних мов;
– підтримку різних типів ліцензійних номерів;
– можливість створення власних LPR-систем.

Platerecognizer використовує відкритий алгоритм розпізнавання номерних знаків, який забезпечує високу точність. Платформа також пропонує широкий спектр налаштувань, які дозволяють розробникам адаптувати її до своїх конкретних потреб.

Користувач системи завантажує потрібне йому зображення з номером машини і отримує повну інформацію про номер машини, регіон, тип транспортного засобу, марку машини та навіть колір машини і її орієнтацію в просторі.

Головною перевагою цієї платформи є те, що вона безкоштовна. Це робить її доступною для кожного. Також до переваг можна віднести підтримку великої кількості мов та ліцензійних номерів.

З недоліків можна виділити складність платформи у використанні та обмежену підтримку зі сторони користувачів та розробників.

На рисунку 1.1 зображено інтерфейс сервісу Platerecognizer.

Рисунок 1.1 – Platerecognizer

Підсумовуючи, можна сказати, що дана система підходить для великої кількості задач. Platerecognizer можна використовувати для моніторингу дорожнього руху, здійснювати контроль за дотриманням правил дорожнього руху, також система підходить для контролю доступу до приватних паркових місць або обмежених територій. Проте на мій погляд система виглядає перевантаженою і тому не підходить для реалізації нашої інформаційної системи.

1.3.2 OpenALPR

OpenALPR – це LPR-платформа, яка заснована на однойменній бібліотеці для автоматичного розпізнавання номерних знаків, написана мовою C++. Програмне забезпечення поширюється, як у комерційній, так і в відкритій версії.

OpenALPR використовує бібліотеки OpenCV і Tesseract OCR. Його можна запускати, як утиліту командного рядка, окрему бібліотеку або фоновий процес, та встановлювати локально на власній системі. Програмне забезпечення також інтегрується з системами керування відео (VMS), такими як Milestone XProtect.

OpenALPR пропонує такий набір функцій:

– підтримку різних типів камер;

– підтримку різних мов;

– є можливість обробки зі звичайних зображень чи з відеопотоку.

На рисунку 1.2 зображено інтерфейс сервісу OpenALPR.

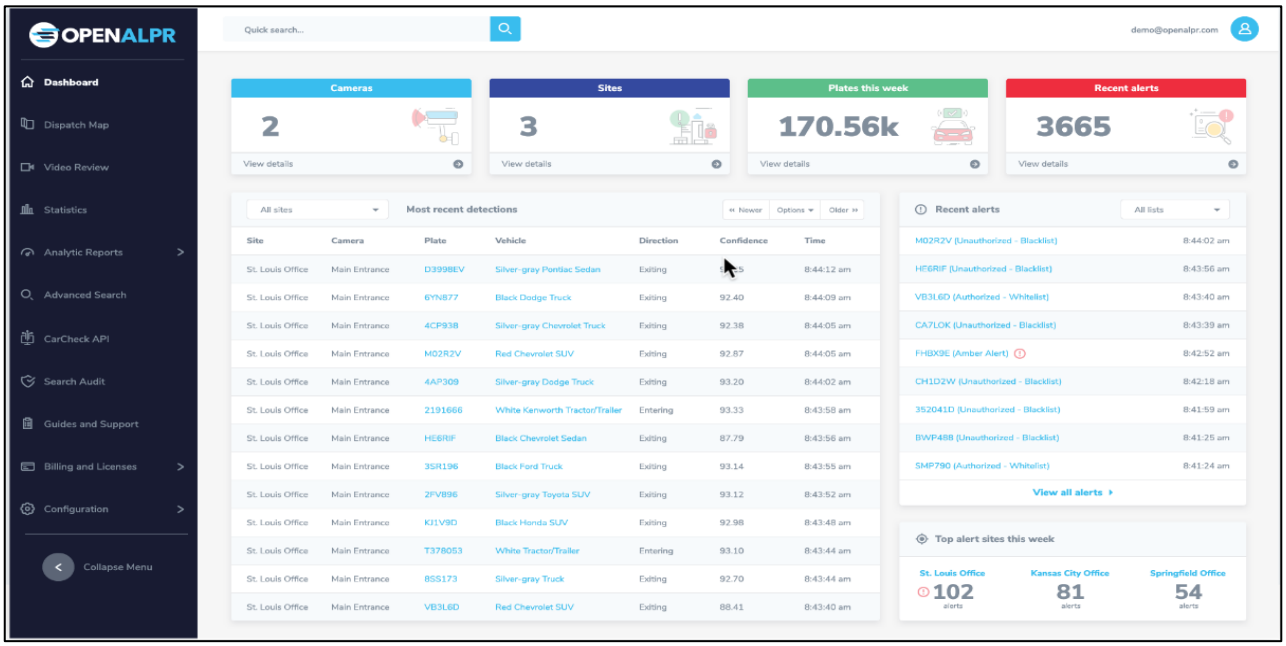

Рисунок 1.2 – OpenALPR

З переваг OpenALPR можна назвати те, що він частково безкоштовний, має велике розмаїття функцій такі, підтримку великої кількості регіонів ліцензійних номерів в тому числі і українських, може зберігати дані про номерні знаки та транспортні засоби до 60 днів та надає можливість створювати власні LPR-системи, що адаптовані до конкретних потреб.

До недоліків можна віднести нижчу точність розпізнавання у порівнняні с конкурентами. OpenALPR має проблеми з розпізнаванням номерних знаків на зображеннях з низькою якістю або зображеннях, які занадто брудні або розмиті.

Також OpenALPR не може впоратися з розпізнаванням номерних знаків з нестандартними форматами або символами. Деякі функції можуть бути складними у налаштуванні, особливо для користувачів, які не мають досвіду роботи з машинним навчанням. Крім того OpenALPR не є повністю безкоштовною, базовий план коштує від 10 доларів.

Узагальнюючи можна сказати, що OpenALPR – це гарна LPR-платформа з великим розмаїттям функцій. Має гарну точність розпізнавання та багато можливостей для різноманітного моніторингу. Але нажаль через відсутність повністю безкоштовного плану використання OpenALPR не підходить в якості LPR-платформи для нашої інформаційної системи.

1.3.3 AWS Rekognition

AWS Rekognition – це хмарне програмне забезпечення від компанії Amazon, яке пропонує технологію комп’ютерного бачення , яка була запущена у 2016 році. Технологія використовується низкою державних установ США, наприклад імміграційною та митною службою, поліцією в декількох штатах Америки, та деякими приватними компаніями [1].

AWS Rekognition дозволяє використовувати можливості комп’ютерного зору, які можна умовно розділити на дві категорії: певні алгоритми, які були попередньо навчені на зібраних даних та алгоритми, які користувач може навчити самостійно на спеціальному наборі даних.

В цілому AWS Rekognition надає такі функції:

– розпізнавання знаменитостей на зображеннях;
– можливість виявлення певних атрибутів обличчя на зображеннях, зокрема можна визначити стать, вік, емоції людини, чи є на волосся (вуси та борода) та чи є на обличчі окуляри і тому подібне;
– виявлення та класифікація тексту, яке знаходиться на зображеннях;
– функція SearchFaces, яка дозволяє користувачам імпортувати базу даних

з зображеннями, на яких вже позначені обличчя та навчати модель штучного інтелекту на цих зображення і в подальшому використовувати цю модель як хмарний сервіс з API.

До переваг AWS Rekognition можна віднести високу точність, швидкість розпізнавання в реальному часі, великий набір різних можливостей, легкість використання та завантаження до тисячі зображень та відеозаписів на місяць безкоштовно протягом 12 місяців.

З недоліків можна виділити необхідність використання хмарних сервісів, щоб мати доступ до застосунків Amazon.

На рисунку 1.3 зображено вигляд сайту AWS Rekognition

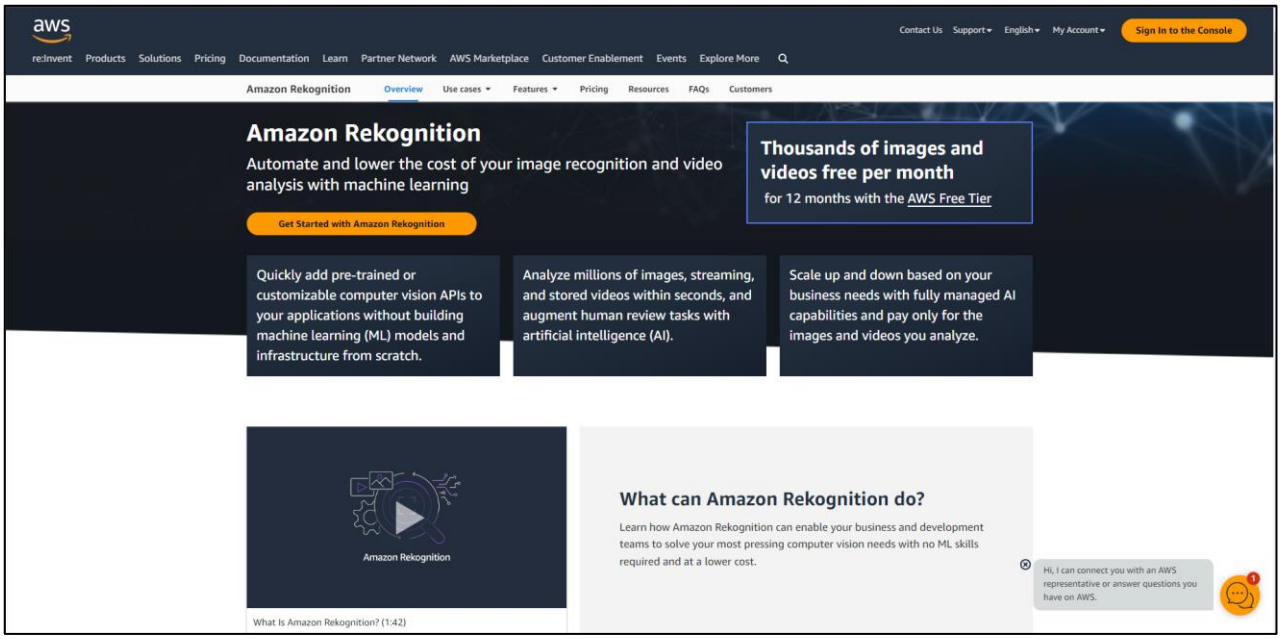

Рисунок 1.3 – AWS Rekognition

В підсумку, можна сказати, що AWS Rekognition – це могутній хмарний сервіс, який може запропонувати велику точність розпізнавання та ціле розмаїття функції від розпізнання звичайного тексту до розпізнавання людей та їх емоцій. Його можливо інтегрувати в інші системи та адаптувати до своїх конкретних потреб. І він цілком підходить під наші задачі, тому можна вважати AWS Rekognition чудовим сервісом для розпізнавання номерних знаків.

1.3.4 Google Cloud Vision API

Google Cloud Vision API – це хмарний сервіс від компанії Google, який пропонує потужні попередньо навчені моделі машинного навчання, для аналізу зображень і знаходженню на них різних об’єктів від звичайних текстів, машин, до людей та їх облич. Клієнтами Google Cloud Vision API є популярна в світі газета The New York Times, компанія Box, компанія eBay та багато інших. Google Cloud Vision API також, як і сервіси описані вище підходить для транспортного управління, безпеки та спостереження.

На рисунку 1.4 зображено приклад роботи сайту Google Cloud Vision API.

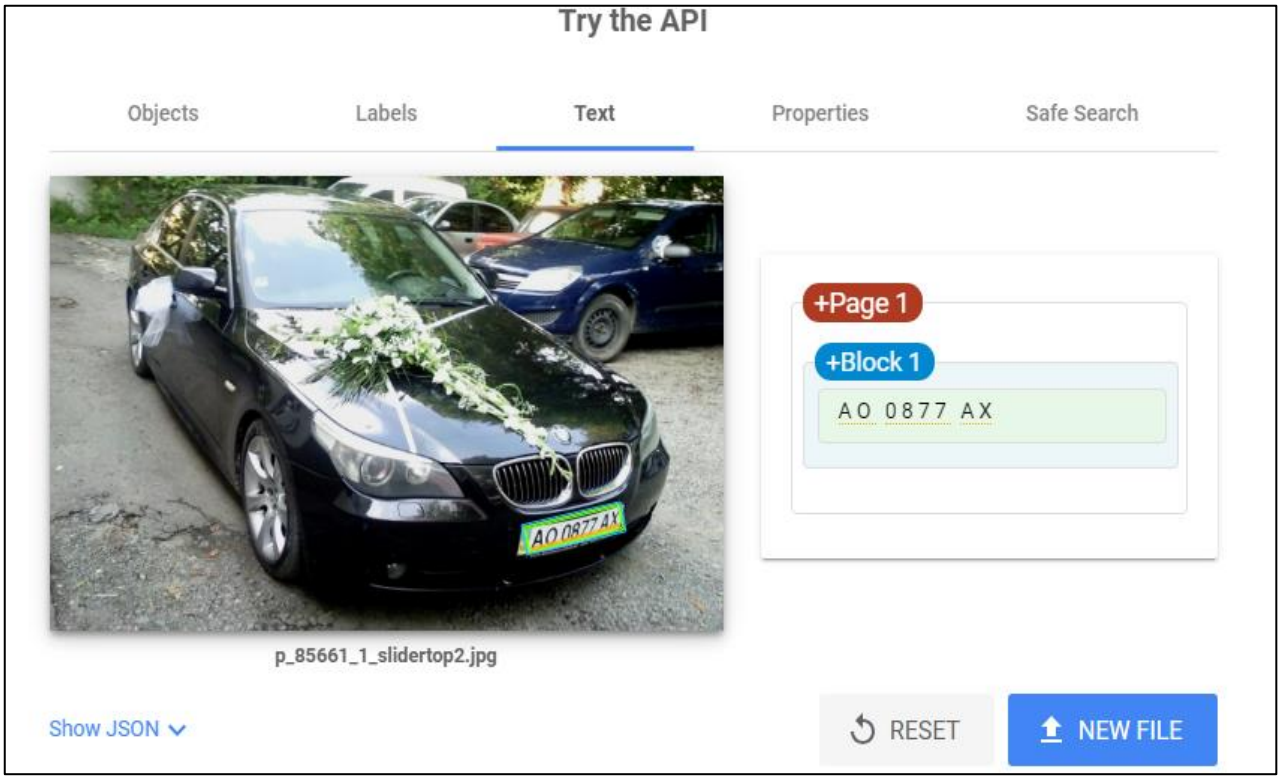

Рисунок 1.4 – Google Cloud Vision API

Google Cloud Vision API має деякі переваги, а саме попередньо навчені моделі штучного інтелекту, які можна використовувати для різних задач. Це значно спрощує роботу розробникам, так як не потрібно створювати і навчати власну модель. Також Google Cloud Vision API доволі легкий у використанні завдяки добре задокументованому інтерфейсу. Додатково потрібно відмітити

доволі зручну та вигідну цінову політику сервісу, розробники платять тільки за функції, якими вони користуються.

Серед негативних сторін можна виділити те, що хоч Google Cloud Vision API дає можливість навчання моделей, він потребує значного досвіту в цій сфері та деяких ресурсів для досягнення гарних результатів. Це стає обмеженням для розробників, які не мають належних навичок в сфері машинного навчання. Також розробники мають доволі обмежений контроль над базовими моделями, що обмежує їхню здатність оптимізувати та адаптувати API до конкретних завдань.

Отже, Google Cloud Vision API – це потужний хмарний сервіс, який пропонує великий інструментарій для отримання інформації з зображень та відео. Однак його технічні недоліки не дозволяють обрати його в якості допоміжної служби в нашій інформаційній системі.

1.3.5 Microsoft Azure AI Vision

Microsoft Azure AI Vision – це хмарний сервіс від компанії Microsoft , яка пропонує інноваційні можливості комп’ютерного зору. В Azure AI Vision є можливість аналізувати зображення, читати текст і виявляти обличчя за допомогою заздалегідь створених тегів до зображень, вилучення тексту за допомогою оптичного розпізнавання символів (OCR) і розпізнавання обличчя. Крім того Microsoft Azure AI Vision вміє виконувати просторовий аналіз для розуміння присутності людей в реальному часі.

Перевагами Microsoft Azure AI Vision є наявність попередньо навчених моделей для різних завдань, що безумовно полегшує роботу, також сервіс легко можна об’єднати з іншими продуктами Azure, що дозволяти створювати потужні додатки. Також до переваг можна віднести те, що розробники платять лише за ресурс, яким вони користуються, що робить Azure AI Vision економічно вигідним варіантом для малих та великих проєктів.

До недоліків можна віднести обмежене налаштування попередньо навчених моделей, вони пропонують обмежений контроль над архітектурою моделі та навчальними даними, що перешкоджає адаптації API до вузькоспеціалізованих завдань.

На рисунку зображено 1.5 приклад роботи сайту Microsoft Azure AI Vision.

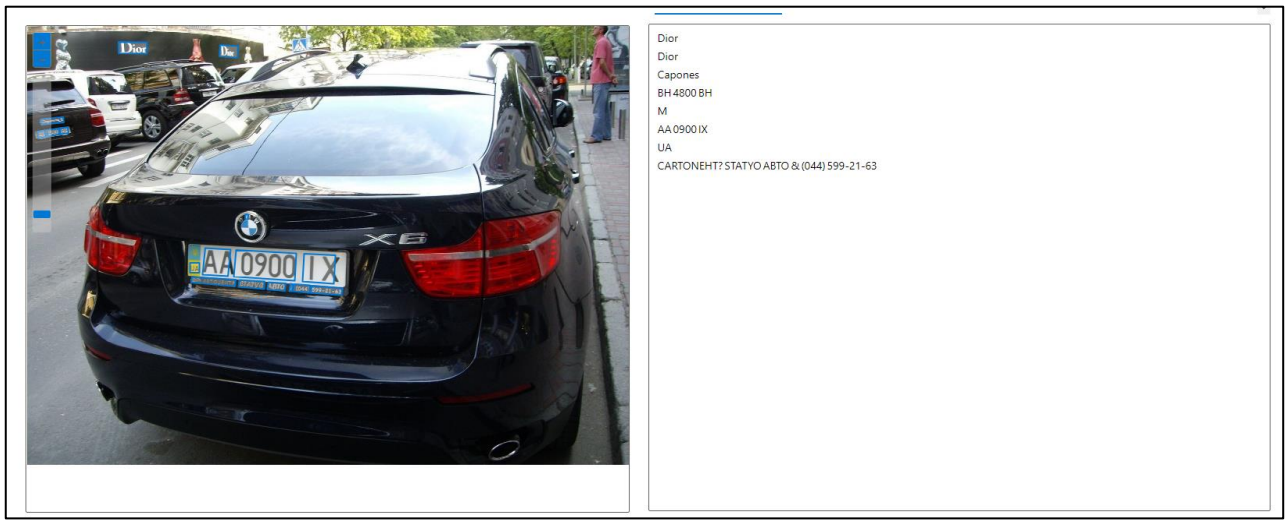

Рисунок 1.5 – Microsoft Azure AI Vision

Роблячи висновок стосовно Microsoft Azure AI Vision можна сказати, що вона схожа на Google Cloud Vision API за своїм функціоналом, який вона надає користувачам, тому і має схожі з нею недоліки, та в результаті не може бути обрана в якості допоміжної служби для нашої інформаційній системі.

1.3.6 Висновки до підрозділу 1.3

Отже, було розглянуто 5 сервісів для розпізнавання тексту, об’єктів та навіть людей. Переваги і недоліки кожного сервісу були розглянуті і описані в відповідних підрозділах. Основними перевагами всіх сервісів було більш-менш високий рівень розпізнавання тексту, легкість в налаштуванні та впровадженні в проєкти. До недоліків можна віднести цінову політику сервісів, та відсутність гнучкості в налаштуваннях для специфічних задач.

І все ж таки приймаючи рішення, який сервіс для розпізнавання тексту буде впроваджено в нашу інформаційну систему «Автомобільна парковка», було обрано сервіс від компанії Amazon під назвою AWS Rekognition. Він повністю відповідає нашим вимогам. Сервіс легкий в налаштуванні та розгортанні, має безкоштовний план використання, якого нам цілком вистачить на період розробки та володіє великою кількістю функцій, які в майбутньому можуть знадобитися, якщо ми забажаємо покращити нашу інформаційну систему.

1.4 Технології та інструменти

Для розробки фронтенд частини нашого застосунку ми використаємо популярний фреймворк від компанії Google під назвою Angular, він ідеально підходить для створення односторінкових вебпрограм.

В якості бекенд частини програми було обрано фреймворк NestJS, бо він ідеально підходить для створення фулстек застосунків в парі з Angular, так як має схожу з ним структуру.

Для зберігання зареєстрованих користувачів будемо використовувати базу даних MongoDB.

В якості сервісу для розпізнавання тексту було обрано AWS Rekognition. Його легкість в використанні та розмаїття функцій ідеально нам підходить.

Інформаційна система «Автоматизованої парковки» має на меті забезпечити легку та ефективну модерацію. Розробка цього застосунку дозволить отримати практичний досвід у розробці інформаційної системи з використанням Angular, NestJS та сервісу AWS Rekognition.

2 ПРОЄКТУВАННЯ СИСТЕМИ

Переходимо до одного з ключових етапів розробки нашої системи, а саме до проєктування системи. Тут ми будемо будувати різні види діаграм, які нам допоможуть краще розробити структуру майбутнього додатку та покращити нашу загальну продуктивність. Для побудови діаграм буде використано онлайн сервіс Draw.io.

Draw.io – це спеціальне онлайн середовище для створення діаграм. Воно підтримує створення блок-схем, UML-діаграм, мережевих діаграм, організаційних діаграм, структурних схем та багато іншого.

2.1 Діаграма прецедентів

Діаграма прецедентів або як ще її називають діаграма варіантів використання – це графічне зображення можливих взаємодій користувачів (акторів) з системою. Діаграма прецедентів демонструє різні варіанти використання системи, типи користувачів системи і нерідко доповнюється іншими типами діаграм.

Діаграма прецедентів допомагає в розумінні функціональності застосунку, допомагає розробникам визначити пріоритети функцій і спроєктувати систему так, щоб вона відповідала потребам користувачів. Також діаграми варіантів використання слугують стислим і вичерпним записом функціональних можливостей системи, що допомагає в майбутньому обслуговуванні та вдосконаленні, крім того цей тип діаграм є цінним інструментом для документування системи та зручності використання.

До ключових елементів діаграми прецедентів належать або актори, варіанти використання та зв’язки. Актори поділяються на первинних та вторинних. Первинні ініціюють варіант використання і є незалежними. Вторинні

актори використовуються системою, проте самостійно взаємодіяти з нею не можуть. Дійові особи в діаграмі переважно зображуються у вигляді паличок. Варіанти використання являють собою функціональні можливості, які пропонуються системою. Варіанти використання зображуються колами або еліпсами. Зв'язки – це лінії, що з'єднують акторів і варіанти використання, вони вказують, як саме актори взаємодіють з функціями системи. Їх поділяють на зв'язки асоціації, зв'язки включення, зв'язки розширення та зв'язки узагальнення. В цілому загальні властивості варіантів використання можна подати в 3 способи, а саме за допомогою зв'язків включення, розширення і узагальнення.

На рисунку 2.1 зображено діаграму прецедентів для інформаційної системи «Автомобільна парковка».

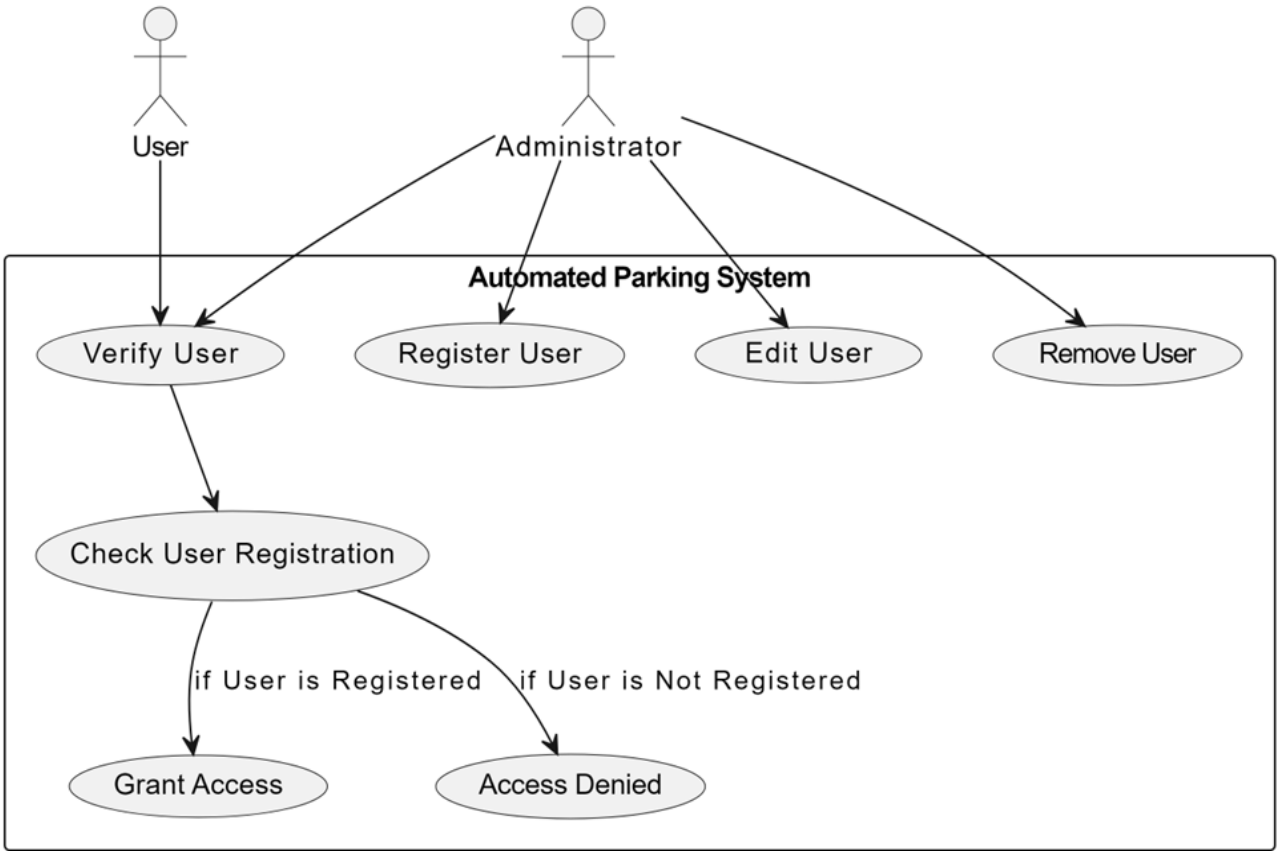

Рисунок 2.1 – Діаграма прецедентів «Автомобільна парковка»

На цій діаграмі ми можемо побачити, як саме відбувається робота нашої системи. Дійова особа «User» ініціює наш перший прецедент, прецедент

верифікації «Verify User». Другою дійовою особою є актор «Administrator», в його функції входить верифікація в системі користувачів, реєстрація нових користувачів, редагування інформації про користувачів та видалення цих самих користувачів. Після активації прецеденту верифікації в дію вступає наступний прецедент «Check User Registration». Він, як стає зрозуміло з назви, перевіряє чи зареєстрований в нашій системі конкретно цей користувач, котрий ініціював верифікацію. Якщо прецедент умовно відповідає «Так», то доступ буде дозволено, про що свідчить прецедент «Grant Access». В іншому випадку, якщо користувача не було знайдено в нашій системі, тобто він не зареєстрований в ній, в такому випадку доступ буде заборонено і відповідний прецедент «Access Denied» буде приведено в дію. Ось як саме структурно буде працювати наша інформаційна система «Автомобільна парковка».

2.2 Діаграма послідовності

Діаграма послідовності – це один з видів UML діаграм, які відображають процеси та задіяні в цих процесах об’єкти, а також послідовність повідомлень, якими вони обмінюються для виконання функціональності.

Вона фокусується на порядку спілкування процесів між собою і допомагає зрозуміти потік управління в рамках окремо взятого сценарію. Це дозволяє розглядати будь-яку взаємодію, як послідовний процес з певними кроками та послідовностями дій.

Діаграма послідовності зосереджується на часовій послідовності подій, демонструючи коли та в якому порядку об’єкти взаємодіють між собою. Завдяки цьому порядок виконання операцій та обмін повідомленнями буде більш точний.

Даний вид діаграм є корисним інструментом для проведення аналізу системи на етапі проєктування, дозволяючи розібратися у взаємодії між компонентами системи та уточнити їхню поведінку. Також її використовують для створення документації проєкту та для комунікації між розробниками,

менеджерами, та замовниками, допомагаючи уточнити та узгодити вимоги до системи.

Перейдемо до основних позначень діаграми послідовності. Паралельними вертикальними лініями, або як їх ще називають лініями життя показано різні процеси або об’єкти, які існують одночасно, а горизонтальними стрілками показано повідомлення, якими вони обмінюються між собою, в порядку їх виникнення.

Типи повідомлень поділяються на синхронні, асинхронні та самоповідомлення. Також на діаграмі у вигляді прямокутників зображують період активності об’єктів на лінії життя. У якості додаткових позначень використовують цикли, певні умови для виконання якоїсь дії та значення повернення. Все це дозволяє більш детально описувати прості та складні сценарії виконання у графічному вигляді.

Використання діаграми послідовності при розробці виявляє потенційні проблеми проєкту, оптимізує послідовність операцій, фіксує поведінку системи для майбутнього обслуговування та модернізації, дозволяють легше розуміти та аналізувати складні взаємодії між об’єктами. Крім того вона використовуються для розробки тестових сценаріїв та тестування функціональності системи, сприяючи виявленню потенційних проблем та помилок у взаємодії між компонентами.

Діаграма послідовності дозволяє узагальнити різноманітні сценарії взаємодії та показати їх загальну структуру, що спрощує розуміння та управління системою. Вона дає можливість моделювати альтернативні варіанти дій у випадку помилок чи виникнення непередбачуваних ситуацій.

Отже, можна сказати, що діаграма послідовності є потужним інструментом, який можна використовувати для візуалізації взаємодій між об’єктами в системі.

На рисунку 2.2 зображено діаграму послідовностей функції верифікації в інформаційній системі «Автомобільна парковка».

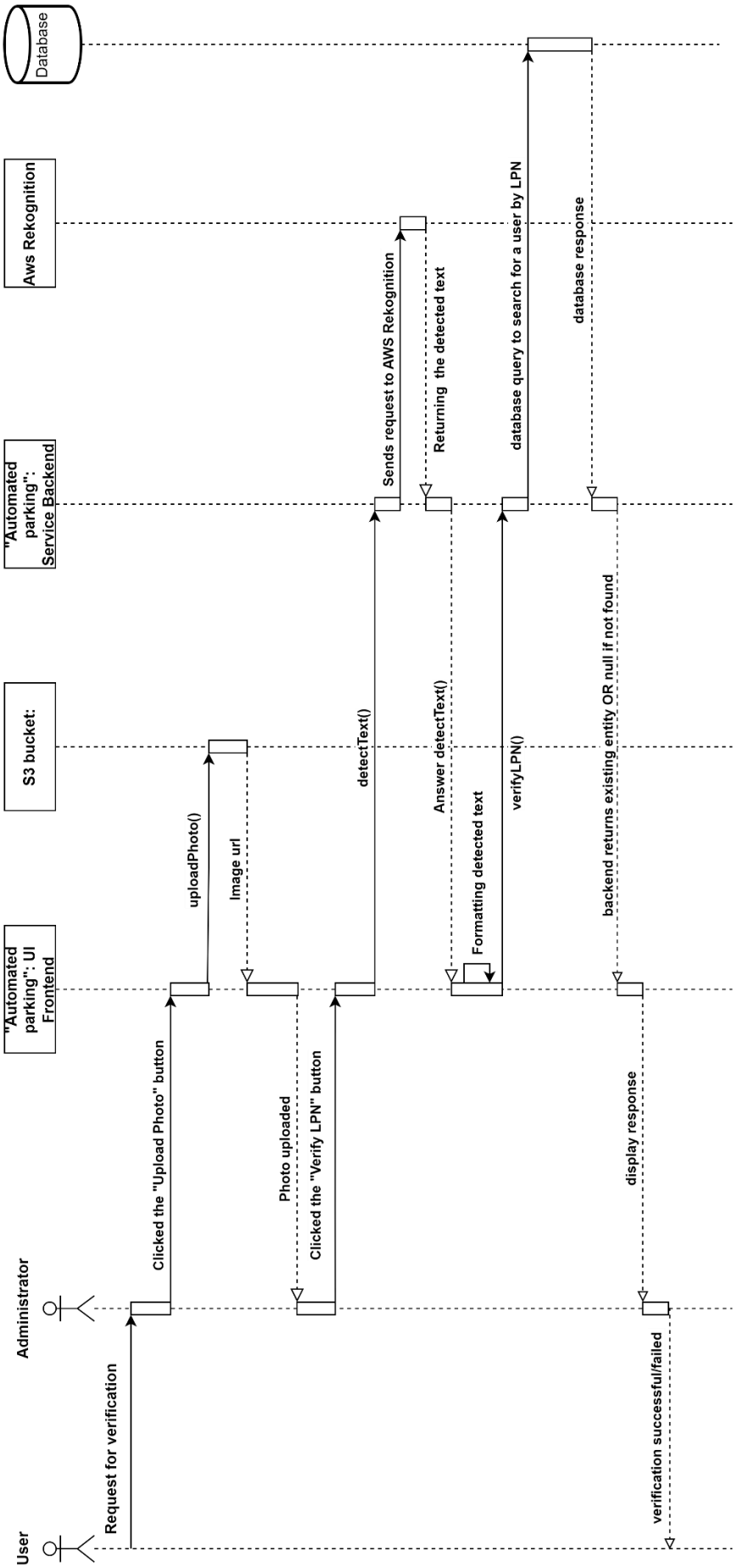

Рисунок 2.2 – Діаграма послідовностей верифікації в системі «Автомобільна парковка»

Перейдемо до пояснення, що ж саме відбувається на нашій діаграмі послідовності. Користувач ініціює запит на верифікацію. Повідомлення про верифікацію надходить до адміністратора, він в свою чергу натискає на кнопку «uploadPhoto» і запускає функцію, яка відправляє фото номерного знаку автомобіля на хмарне сховище даних під назвою Amazon Simple Storage Service або як його ще називають Amazon S3. Сервіс надає можливість для зберігання та отримання будь-якого обсягу даних. Він повертає нам посилання на завантажену картинку.

Далі адміністратор натискає кнопку «Verify LPN» і запускає функцію «detectText», яка передає отримане посилання на зображення до бекенд частини нашої системи. Потім бекенд робить запит на сервіс для розпізнавання тексту AWS Rekognition.

AWS Rekognition опрацьовує надане зображення та повертає знайдений текст до бекенду, який в свою чергу повертає його до фронтенд частини.

Фронтенд частина форматує знайдений текст, знаходить номер автомобіля та передає його далі у функцію «VerifyLPN». Ця функція передає знайдений номер автомобіля до бекенду, який виконує запит до бази даних, щоб знайти співпадіння серед зареєстрованих номерів автомобілів. І якщо номер автомобіля було знайдено в базі даних, то користувач вважається зареєстрованим в нашій системі і він успішно проходить верифікацію і доступ буде дозволено.

В іншому випадку, якщо номер автомобіля не було знайдено, то користувач вважається не зареєстрованим, а отже верифікація буде безуспішна і доступ буде заборонено.

Загалом, ось так структурно буде працювати верифікація в нашій інформаційній системі. Створення діаграми послідовностей для верифікації дозволило детально змоделювати кожен крок цього процесу та в майбутньому виявити можливі помилки. Також діаграма покращила загальне розуміння структури майбутнього фунціоналу в нашій системі і як наслідок покращить основну структуру проєкту та подальшу роботу над ним.

2.3 Дизайн системи

На основі побудованих діаграм в минулих підрозділах, можна уявити приблизний дизайн нашої майбутньої системи.

На рисунку 2.3 зображено макет сторінки верифікації в системі. По середині сторінки знаходиться форма для завантаження фото автомобіля. В формі є дві кнопки. Кнопка для завантаження фото та кнопка для верифікації. Якщо натиснути на кнопку завантаження фото воно буде завантажено на сторінку і його можна буде переглянути, а після натискання на кнопку верифікації можна побачити відповідне повідомлення з результатом верифікації.

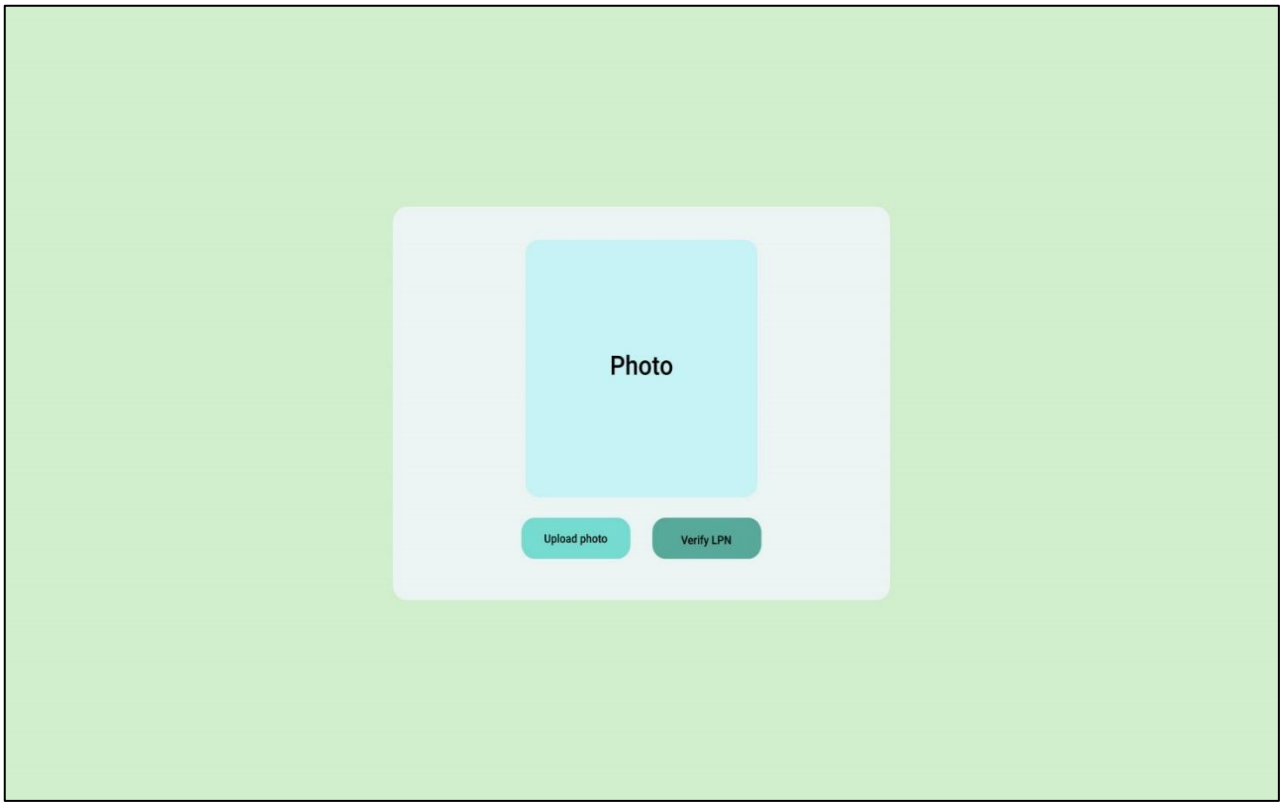

Рисунок 2.3 – Макет сторінки верифікації в системі «Автомобільна парковка»

На рисунку 2.4 наведено макет сторінки реєстрації користувача в системі. Сторінка має невеличкий опис з лівого боку та форму для додавання нового користувача в системі.

В формі є 6 полів, ім’я, прізвище, електронної пошти, дати народження,

номерного знаку автомобіля та паркувального місця. Внизу знаходиться кнопка для реєстрації користувача в системі.

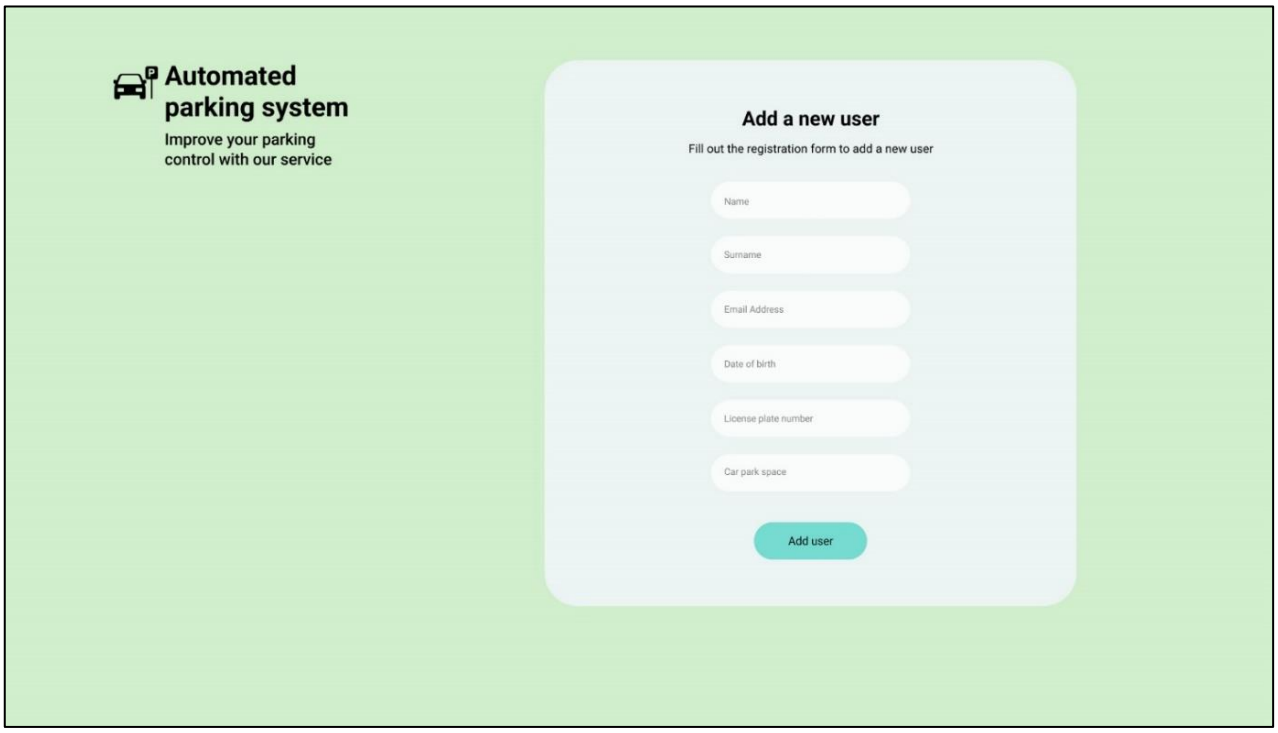

Рисунок 2.4 – Макет сторінки реєстрації в системі «Автомобільна парковка»

На рисунку 2.5 показано модальне вікно, яке з’являється при успішній верифікації.

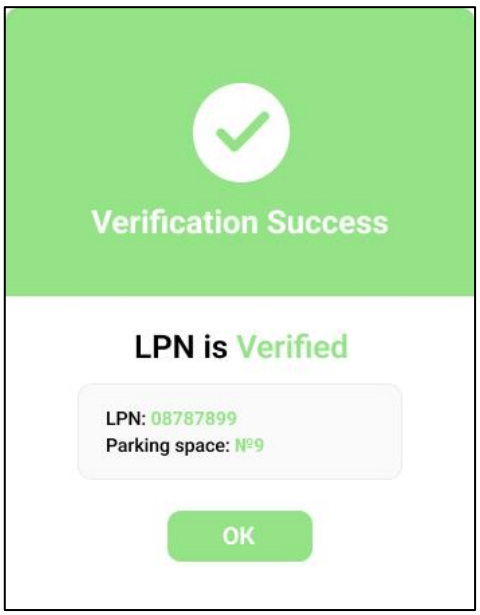

Рисунок 2.5 – Макет сторінки модального вікна при успішній верифікації

На модальному вікні успішної верифікації можна побачити номер автомобіля та номер паркувального місця.

На рисунку 2.6 показано модальне вікно, яке з’являється при безуспішній верифікації.

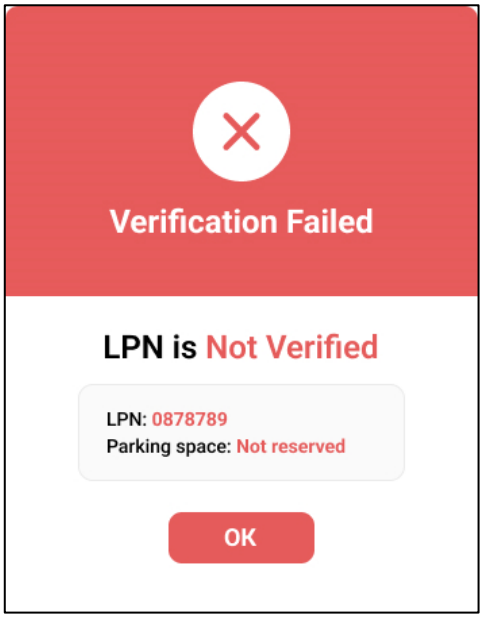

Рисунок 2.6 – Макет сторінки модального вікна при безуспішній верифікації

На модальному вікні безуспішної верифікації можна побачити номер автомобіля та характерний надпис про те, що паркувальне місце за цим номером автомобіля не знайдено.

Загалом, дизайн інтерфейсу повністю відповідає усім вимогам. Всі деталі інтерфейсу знаходяться на видному місці. При успішній або безуспішній верифікації буде з’являтися модальне вікно, яке буде інформувати адміністратора системи.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Опис інструментів розробки

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

Angular – це популярний фронтенд фреймворк [2], який базується на мові програмування Typescript [3]. Підходить для створення динамічних і складних вебдодатків. Angular пропонує великий набір функцій та інструментів, які полегшують розробку інтерфейсу та забезпечують структуровану і передбачувану структуру додатку. Також Angular ідеально підходить для створення односторінкових додатків [4], де вміст динамічно завантажується без оновлення сторінки, пропонуючи більш плавний користувацький досвід.

NestJS – це платформа для створення ефективних, масштабованих програм на стороні серверу. Використовує мову програмування TypeScript для створення масштабованих та ефективних серверних додатків. NestJS застосовує модульний дизайн, розбиваючи додаток на незалежні модулі. Це сприяє організації коду, полегшує повторне використання та спрощує обслуговування [5].

Також використовується бібліотека для реактивного програмування RxJS та бібліотека готових компонентів інтерфейсу для Angular-додатків.

3.2 Angular-компонент

Angular-компоненти використовуються для створення інтерфейсів в Angular-додатках. Компоненти інкапсулюють певну функціональність, поведінку та елементи інтерфейсу. Що забезпечує їх багаторазове використання в різних частинах програми. В компоненті зберігаються його HTML-шаблон, файл стилів CSS та файл TypeScript, який зберігає в собі логіку компоненту.

Наведемо приклад файл TypeScript з компоненту сторінки реєстрації користувача в системі (рис. 3.1).

```
export class AdminPageComponent implements OnInit {
  fg = new FormGroup({
    name: new FormControl('', Validators.required),
    surname: new FormControl('', Validators.required),
    LPN: new FormControl('', Validators.required),
    email: new FormControl('', Validators.required),
    model: new FormControl('', Validators.required),
    parkspace: new FormControl('', Validators.required),
  })

  errorText: string = '';

  constructor(private appService: AppService) {

  }
  ngOnInit(): void {
  }

  CreateUserCheck() {
    if(this.fg.invalid) {
      this.errorText = "Please fill in all required fields.";
    }
    const bodyUser = this.fg.value as LPN;
    const finalLPN: string = bodyUser.LPN.replaceAll('А', 'A').replaceAll('В', 'B').replaceAll('С', 'C')
    .replaceAll('Е', 'E').replaceAll('Н', 'H').replaceAll('І', 'I').replaceAll('К', 'K')
    .replaceAll('М', 'M').replaceAll('О', 'O').replaceAll('Р', 'P').replaceAll('Т', 'T').replaceAll('Х', 'X');
    this.appService.createUser({
      ...bodyUser,
      LPN: finalLPN
    }).subscribe(data => console.log(data))
  }
}
```

Рисунок 3.1 – Приклад файл TypeScript з компоненту сторінки реєстрації користувача

Повний код файлу наведено в Додатку А.

3.3 Angular-інтерфейс

Angular-інтерфейс – своєрідна інструкція, яка використовується для визначення структури об’єкта [8]. Це сприяє ясності та узгодженості коду, оскільки визначає, які саме властивості або методи має мати об’єкт певного типу. вони задають очікувану структуру даних та методів, що спрощує спілкування між різними частинами коду, зменшує ймовірність помилок. Крім того,

використання інтерфейсів робить код більш читабельним та підвищує його сумісність та масштабованість.

Наведемо приклад інтерфейсу для об’єкту користувач (рис. 3.2).

```
export interface LPN {
  name?: string;
  surname?: string;
  email?: string;
  LPN: string;
  model?: string;
  parkspace?: string;
  createdAt?: string;
  updatedAt?: string;
  status?: boolean;
}
```

Рисунок 3.2 – Приклад інтерфейсу для об’єкту користувач

3.4 Angular-сервіс

Angular-сервіс – це певний клас, який зберігає в собі багаторазову логіку, до якої можна отримати доступ і спільно використовувати в різних компонентах додатку. Він надає можливість відокремити певний функціонал від компонентів [9].

Angular-сервіси можуть включати у себе різні функції. Можна отримувати дані з сервера через HTTP-запити, реалізації бізнес-логіку системи, перевірку даних, роботу з кешем та багато іншого. Вони є важливим елементом створення додатків, оскільки дозволяють використовувати один і той же функціонал у різних частинах програми без дублювання коду.

За допомогою Angular Dependency Injection, сервіси можна інжектити у компоненти чи інші сервіси, щоб вони могли використовувати його функціонал. Це спрощує взаємодію між компонентами та дає можливість створювати зручні та підтримувані застосунки. Angular-сервіси підтримують чистоту компонентів, розділяючи їхню логіку від повторюваного коду.

Наведемо приклад Angular-сервісу (див. рис. 3.3).

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { environment } from 'src/environments/environment';
import { HttpClient } from '@angular/common/http';
import { LPN } from 'src/app/components/admin-page/admin-page.component';

@Injectable({
  providedIn: 'root'
})
export class AppService {
  Api_url: string = environment.API_URL


  constructor(private httpClient: HttpClient) { }

  getLink(file: File): Observable<{ url: string }> {
    return this.httpClient.post<{ url: string }>(
      `${this.Api_url}/upload/link`,
      { filename: file.name, size: file.size }
    );
  }

  uploadImage(url: string, file: File | undefined) {
    if (!file) {
      return of();
    }
    return this.httpClient.put(url, file);
  }

  detectText(image: string): Observable<{texts: string[]}> {
    return this.httpClient.post<{ texts: string[] }>(
      `${this.Api_url}/detect`,
      { image : image }
    );
  }
```

Рисунок 3.3 – Приклад Angular-сервісу

Повний код сервісу наведено в Додатку Б.

3.5 NestJS-контролери

NestJS-контролери – це клас, який відповідає за обробку вхідних запитів, взаємодіючи з різними частинами додатку та визначаючи, як саме будуть оброблятися ці запити. Вони використовується для визначення маршрутів,

обробки даних запиту та генерування відповідей.

Вони забезпечують чіткий розподіл обов'язків. Контролери відповідають за обробку HTTP-запитів, а інші частини додатку, такі як сервіси, відповідають за логіку бізнесу та доступ до даних.

Контролери можна повторно використовувати в різних частинах додатку, просто ін'єктувавши в них різні служби та дані.

Наведемо приклад контролеру (рис. 3.4).

```
import { Body, Controller, Post, UnauthorizedException } from '@nestjs/common';
import { LpnService } from './lpn.service';
import { CreateLpnDto } from './dto/lpn.dto';

@Controller('lpn')
export class LpnController {

    constructor(private readonly lpnService: LpnService) {}

    @Post('create')
    async createUser(@Body() body: CreateLpnDto) {
        console.log('BODY: ', body);
        return await this.lpnService.createLpn(body);
    }

    @Post('verify')
    async getLpn(@Body() body: {LPN: string}) {
        console.log('BODY: ', body);
        if (!body?.LPN) {
            throw new UnauthorizedException('LPN cannot be empty');
        }
        return await this.lpnService.getLpn(body);
    }
}
```

Рисунок 3.4 – Приклад NestJS-контролерів

3.6 Реалізація верифікації

Головною функцією в нашій системі буде верифікація користувача в системі. Для її реалізації ми будемо використовувати усі перелічені вище технології та засоби.

Спочатку створимо компонент, в якому буде знаходитися функція верифікації. Для створення компоненту в Angular виконуємо команду «ng generate component» і кінці пишемо назву свого компонету [6].

Після того як компонент буде згенеровано можна перейди до написання HTML частини нашого компоненту за макетом, який був зображений в попередньому розділі.

На рисунку 3.5 зображено HTML розмітку сторінки для верифікації.

```
<div class="w-full flex justify-center">
  <div class="w-[750px] h-[535px] bg-[#ECF4F3] rounded-2xl
  flex items-center pt-11 gap-7 flex-col" >
    <div *ngIf="!base64Image" class="w-[350px] h-[350px] bg-[#C6F3F5]
    rounded-2xl flex items-center justify-center font-medium text-4xl">
      Photo
    </div>
    <img [src]="base64Image" *ngIf="base64Image" alt="Photo" class="w-auto
    h-[350px] object-cover">
    <div class="flex gap-8">
      <label class="input-file">
        <input type="file" name="UploadImage"
        (change)="onFileChange($event)" accept="image/*">
        <span>Upload Photo</span>
    </label>
      <button (click)="detectText()" class="w-40 h-14 rounded-2xl bg-[#f5a879]
      hover:bg-[#6cd1be] text-base font-medium">Verify LPN</button>
    </div>
  </div>
</div>
```

Рисунок 3.5 – HTML компоненту верифікації

Поглянемо як виглядає сторінка верифікації в браузері (див. рис. 3.6). Як можна помітити наша сторінка відповідає дизайну з минулого розділу.

Після створення візуальної частини компоненту ми можемо перейти до написання інших потрібних функції.

Згадуючи діаграму послідовності нам спочатку потрібно написати функцію, яка буде завантажувати фотографію номеру автомобіля потрібного для верифікації на хмарне сховище AWS S3 [10].

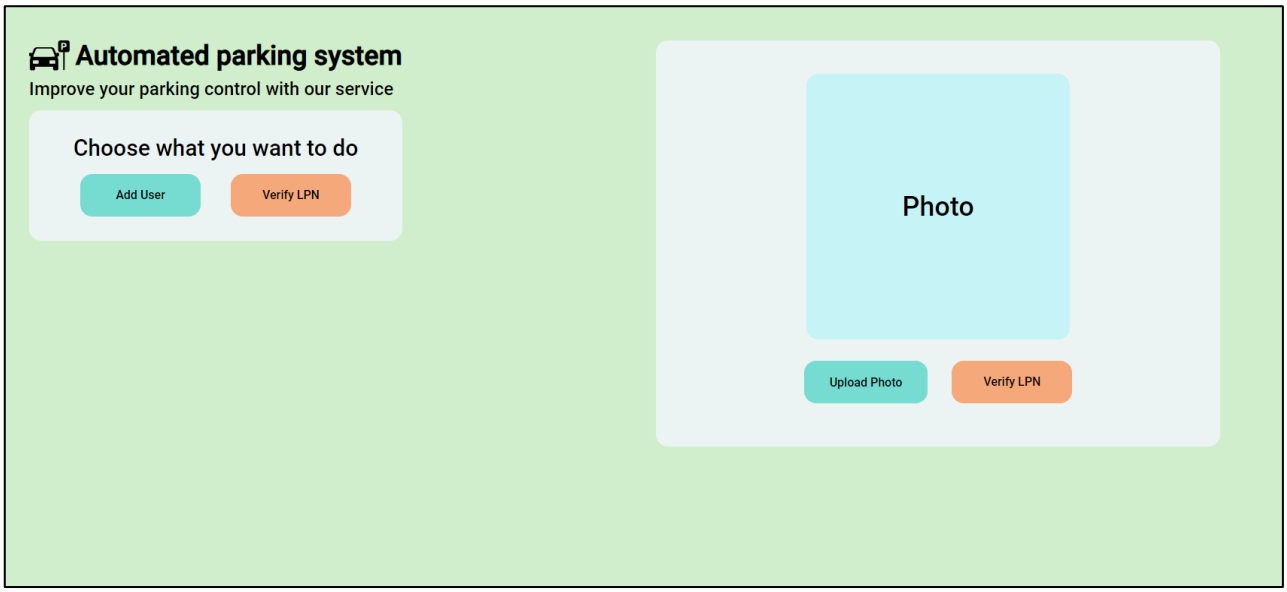

Рисунок 3.6 – Сторінка верифікації

Для того, щоб завантажити файли на хмарне сховище S3 спочатку потрібно отримати підписану URL-адресу. Це зроблено задля уникнення небажаного завантаження файлів, прискорення обміну даних, спрощення процесу інтеграції та для резервування місця для нашого зображення.

Щоб отримати потрібну нам адресу для завантаження файлу першим чином ми створимо функцію, яка буде приймати вхідний файл (завантажений через поле вводу в HTML-шаблоні компоненту) та передавати його до бекенду через функцію getLink (рис. 3.7). Сама функція виклику попередньої виглядає так, як представлено на рисунку 3.8.

Як бачимо, у функції з рисунку 3.8, ми передаємо файл у функцію getLink, підписавшись на цю функцію, ми викликаємо HTTP-запит, результатом якого буде потрібний підписаний URL.

```
getLink(file: File): Observable<{ url: string }> {
  return this.httpClient.post<{ url: string }>(
    `${this.Api_url}/upload/link`,
    { filename: file.name, size: file.size }
  );
}
```

Рисунок 3.7 – Код функції getLink

```
onFileChange(event: any) {
  this.file = event.target.files[0];
  if(!this.file) return
  this.convertToBase64();
  this.appService.getLink(this.file).subscribe((data: {url: string}) => {
    if (!data || !this.file) return;
    this.uploadImg(data.url, this.file);
  })
}
```

Рисунок 3.8 – Код функції onFileChange

Після цього ми передаємо отриманий підписаний URL разом з нашим файлом у функцію uploadImg.

Функція uploadImg має такий вигляд, як наведено на рисунку 3.9.

```
uploadImg(url: string, file: File) {
    const formattedUrl: URL = new URL(url);
    this.imgToSave = formattedUrl.pathname.replace('/', '');
    this.appService.uploadImageToS3(url, file).subscribe((data: any) => {
      this.loading = false;
    });
  };
}
```

Рисунок 3.9 – Код функція uploadImg

Функція форматує підписаний URL, зберігає назву фотографії для подальшої взаємодії з ним та викликає функцію uploadImageToS3.

На рисунку 3.10 зображена функція uploadImageToS3.

```
uploadImageToS3(url: string, file: File | undefined) {
  if (!file) {
    return of();
  }
  return this.httpClient.put(url, file);
}
```

Рисунок 3.10 – Код функція uploadImageToS3

Функція uploadImageToS3 за допомогою HTTP-запиту PUT завантажує наше зображення через підписаний URL на хмарне сховище S3 і тепер воно стає доступне для функції верифікації.

На рисунку 3.11 можна побачити результат виконання усіх функції, які були описані вище. Зображення успішно завантажено і готове до перевірки.

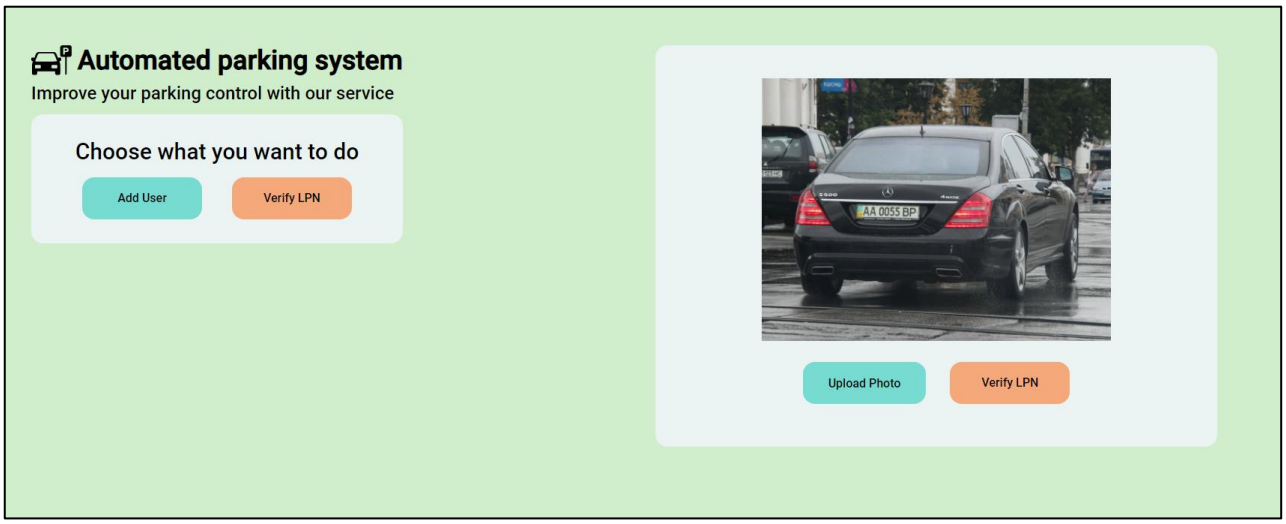

Рисунок 3.11 – Результат роботи функції uploadImg

Тепер можна безпосередньо перейти до розробки функції верифікації. Розпочнемо з функції, яка буде розпізнавати текст, який знаходиться на зображенні. Виглядати вона буде наступним чином (див. рис. 3.12).

```
detectTextfromPicture(image: string): Observable<{texts: string[]}> {
  return this.httpClient.post<{ texts: string[] }>(
    `${this.Api_url}/detect`,
    { image : image }
  );
}
```

Рисунок 3.12 – Код функції detectTextfromPicture

Ця функція приймає в собі назву нашого зображення та виконує запит на серверну частину застосунку, яка повертає нам ідентифікований текст.

На рисунку 3.13 зображено допоміжну функцію detectText, яка отримує результат функції detectTextfromPicture у фронтенд частину додатку, зберігає

його в змінній та викликає функцію обробки цього результату.

```
detectText(){
  this.appService.detectTextfromPicture(this.imgToSave).subscribe((data:{texts: string[]}) => {
    this.textDetected = data.texts;
    this.processLpnResponse();
  })
}
```

Рисунок 3.13 – код функції detectText

На рисунку 3.14 зображено функцію processLpnResponse, яка займається обробкою результату розпізнавання тексту.

```
processLpnResponse() {
  let res = this.textDetected.filter(text => this.PlateNumberRegex.test(text.replaceAll(/ /g, ''))).map(text => {
   return text.replaceAll(/ /g, '').replaceAll('A', 'A').replaceAll('B', 'B').replaceAll('C', 'C')
   .replaceAll('E', 'E').replaceAll('H', 'H').replaceAll('I', 'I').replaceAll('K', 'K')
   .replaceAll('M', 'M').replaceAll('O', 'O').replaceAll('P', 'P').replaceAll('T', 'T').replaceAll('X', 'X')
  });
  if(!res.length) {
    console.log('LPN is not detected');
    return;
  }
  const LPNS = this.getUniqueValues(res);
  this.componeVerifyLpnRequests(LPNS);
}
```

Рисунок 3.14 – код функції processLpnResponse

Для початку нам потрібно відфільтрувати весь текст, для того щоб знайти номер автомобілю. Фільтруємо ми за допомогою регулярного виразу. Після цього ми оброблюємо результат, який нам повернула функція перевірки на відповідність в регулярному виразі.

Далі нам потрібно отримати унікальні номери автомобілів, щоб запобігти повторенню одного і того самого номеру. Цим займається функція getUniqueValues (див. рис. 3.15).

```
getUniqueValues(res: string[]) {
  return [...new Set(res)]
}
```

Рисунок 3.15 – Код функції getUniqueValues

Потім отримані результати передаємо до функції componeVerifyLpnRequests (див. рис. 3.16), яка використовується для перевірки масиву результатів.

```
componeVerifyLpnRequests(lpnsToVerify: string[]) {
  const requests: Observable<LPN | null>[] = lpnsToVerify.map((lpn) => this.verifyLPN(lpn));
  const reqs = forkJoin([...requests]).pipe(
    catchError((err: HttpErrorResponse) => {
      console.log( err.error.message  || err.message );
      this.loading = false;
      return of();
    })
  );
  reqs.subscribe((data: (LPN | null)[]) => {
    console.log(data);
    const success: (LPN| null)[] = data.filter(item => Boolean(item));
    if(success.length) {
      this.openDialog({...success[0]!, status: true})
    }else {
      this.openDialog({LPN: lpnsToVerify[0], status: false})
    }
  })
}
```

Рисунок 3.16 – Код функції componeVerifyLpnRequests

Спочатку функція створює масив Observable, кожен з яких є запитом на перевірку. Запит на перевірку виконується за допомогою функції verifyLPN, повний код якої буде в додатку В. Далі за допомогою оператору бібліотеки RxJS [7] під назвою forkJoin ми об’єднуємо масив Observable в один Observable, який буде емітити об’єднатий результат кожного запиту на перевірку, коли вони всі завершиться.

Далі ми підписуємося на результати перевірки та фільтруємо їх, щоб відібрати тільки успішні перевірки. І далі, якщо є успішні перевірки показуємо модальне вікно успішної перевірки, в іншому випадку показуємо вікно безуспішної перевірки.

На рисунку 3.17 зображено модальне вікно успішної верифікації. Можна побачити, що модальне вікно показує номер автомобілю та паркувальне місце цього автомобіля. На рисунку 3.18 зображено модальне вікно безуспішної верифікації.

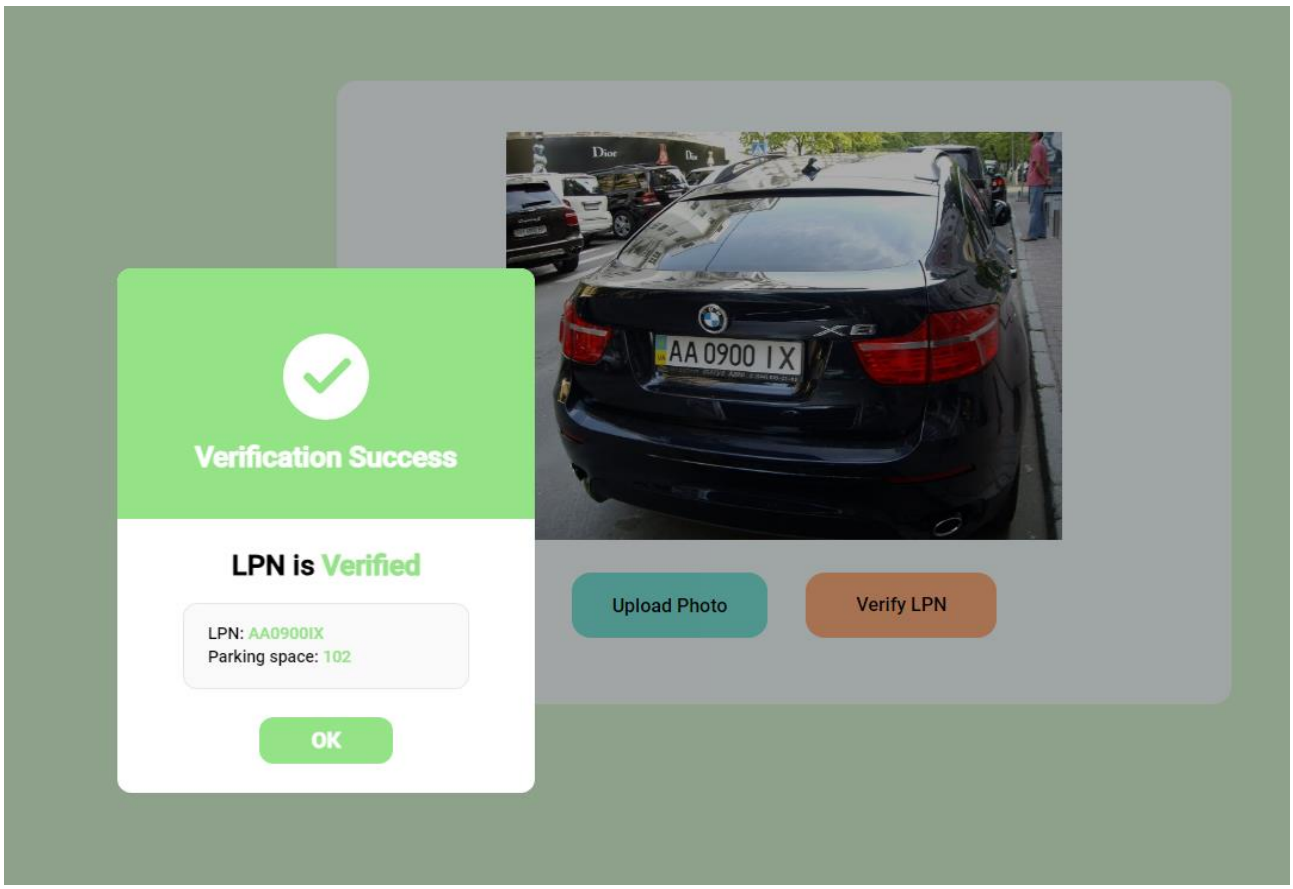

Рисунок 3.17 – Приклад успішної верифікації

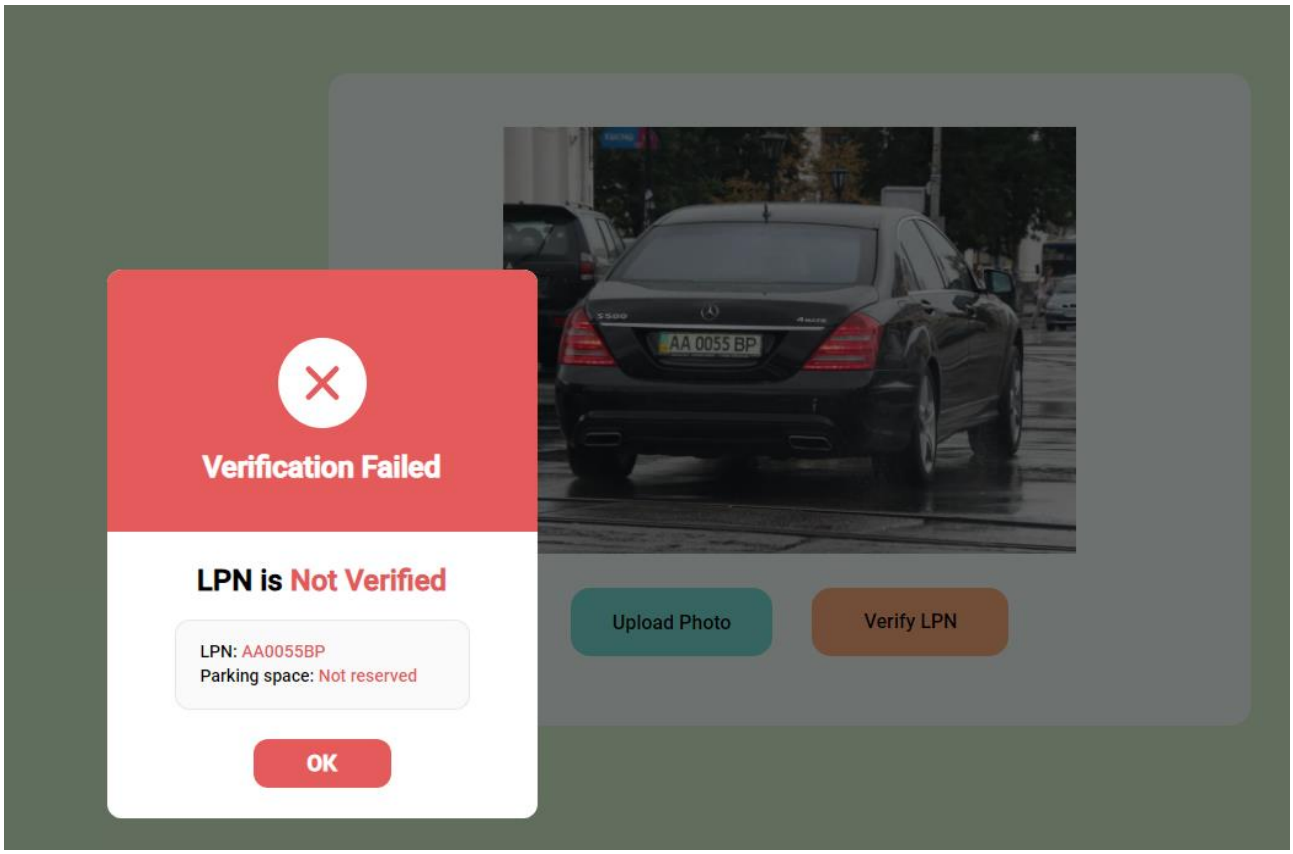

Рисунок 3.18 – Приклад безуспішної верифікації

Загалом можна сказати, що функція верифікації була реалізована і дозволяє здійснювати контроль зареєстрованих користувачів в системі.

3.7 Реалізація серверної частини

3.7.1 Реалізація завантаження файлу на AWS S3

Серверна частина нашого додатку відіграє суттєву роль в роботі нашої інформаційної системи. Для початку роботи над бекендом, нам потрібно створити проєкт NestJS. Для цього в терміналі пишемо команду «nest new project name».

Потім потрібно підключити npm пакети для роботи з AWS S3 та AWS Rekognition. Для AWS S3 потрібно написати команду «npm i --save @aws-sdk/s3-request-presigner», а для AWS Rekognition «npm i --save @aws-sdk/client-rekognition», після цього з сервісами AWS можна працювати.

Також серверна частина потребує підключення бази даних, ми скористаємося MongoDB. Для цього спочатку в терміналі прописуємо команду «npm i @nestjs/mongoose mongoose».

Потім у файлі «app.module.ts» імпортуємо модуль бази даних (див. рис. 3.19).

```
@Module({
  imports: [
    ConfigModule.forRoot({ isGlobal: true }),
    MongooseModule.forRoot(`mongodb+srv://suhanof656:${process.env['MONGODB_PASSWORD']}@parking0.cc0dsab.mongodb.net/?retryWrites=true&w=majority`),
    DetectModule,
    UploadModule,
    LpnModule,
  ],
```

Рисунок 3.19 – Приклад імпорту бази даних MongoDB

Після того, як були виконані основні налаштування серверної частини, можна перейти до створення потрібного функціоналу.

Для початку створимо функцію, яка буде створювати та повертати

фронтенд частині підписане посилання для завантаження зображення для верифікації.

На рисунку 3.20 зображена функція getFileUrl.

```
async getFileUrl(file: {filename: string; size: number}) {
  const bucket: string = this.configService.getOrThrow('AWS_BUCKET_NAME');
  const command = new PutObjectCommand({
          Bucket: bucket,
          Key: file.filename,
          ContentLength: file.size
      });
  const url = await getSignedUrl(this.s3Client, command, { expiresIn: 1200 });
  return {url: url};
 }
}
```

Рисунок 3.20 – Код функції getFileUrl

Спершу функція отримує інформацію про файл, який потрібно завантажити. Інформація береться з об’єкту file, який зберігає ім’я файлу та його розмір. Далі створюється об’єкт PutObjectCommand, який містить інформацію про завантажений файл. Використовуємо функцію getSignedUrl() для генерування попередньо підписаної URL-адреси.

Функція getSignedUrl() повертає об’єкт, в якому одне поле url. Ця властивість містить згенеровану попередньо підписану URL-адресу, яку можна використовувати для завантаження файлу. І в кінці повертаємо URL до фронтенд частини.

Далі потрібно створити функцію завантаження файлу до AWS S3. На рисунку 3.21 зображена функція «uploadFileToS3».

```
async uploadFileToS3(file: Express.Multer.File) {
  const bucket: string = this.configService.getOrThrow('AWS_BUCKET_NAME');
  return this.s3Client.send(
      new PutObjectCommand({
          Bucket: bucket,
          Key: file.fieldname,
          Body: file.buffer,
      })
  )
}
```

Рисунок 3.21 – Код функції uploadFileToS3

На вході функція отримує Express.Multer.File об’єкт, який надає детальну інформацію про завантажений файл. Створюється об’єкт PutObjectCommand, який містить інформацію про завантаження файлу. Потім використовується об’єкт клієнта S3 для відправлення PutObjectCommand до Amazon S3.

Нам повертається результат відправлення команди. І потім він повертається на фронтенд частину.

3.7.2 Реалізація розпізнавання тексту з допомогою AWS Rekognition

Тепер коли функцію для завантаження файлу створено і сервіси AWS зможуть вільно між собою взаємодіяти, перейдемо до реалізації функції розпізнавання тексту.

На рисунку 3.22 зображена функція «detectText».

```
async detectText(
    body: { image: string }
) {
    const input: DetectTextCommandInput = {
        Filters: {
            WordFilter: {
                MinConfidence: 10,
            }
        },
        Image: {
            S3Object: {
                Bucket: this.bucket,
                Name: body.image,
            },
        },
    };
    const command = new DetectTextCommand(input);
    let response: DetectTextCommandOutput | undefined = undefined;
    try {
        response = await this.client.send(command);
    } catch (err) {
        console.log(err);

        throw new HttpException(
            'Error: Something went wrong while trying to detect texts!',
            HttpStatus.BAD_REQUEST,
        );
    }
    console.log(response);
    const texts: string[] = response ? response.TextDetections.map((detection: TextDetection) => detection.DetectedText) : [];
    return {texts: texts };
}
```

Рисунок 3.22 – Код функції detectText

Загалом, цей код виявляє текст на зображенні, яке зберігається в Amazon S3. Отримуваним параметром функції є ім’я зображення з об’єкта body. Створюється об’єкт DetectTextCommand, який містить інформацію про зображення, яке потрібно проаналізувати. відправляє команду DetectTextCommand до Amazon Rekognition. Потім отримуємо результат розпізнавання тексту з Amazon Rekognition, витягуємо текст з результату і об’єкт з текстом відправляємо до фронтенду.

Тепер перейдемо до функції верифікації. Ціль функції полягає в тому, щоб перевірити чи зареєстрований користувач с таким номером автомобіля в нашій системі і повернути його до фронтенду.

На рисунку 3.23 зображена функція getLpn.

```
async getLpn(body: {LPN: string}) {
    const lpn = await this.lpnModel.findOne({ LPN: body.LPN });
    if (!lpn) {
        throw new UnauthorizedException('LPN does not exist');
    }
    return lpn;
}
```

Рисунок 3.23 – Код функції getLpn

Спочатку отримуємо номер автомобіля з об’єкту. Потім робимо пошук номеру автомобіля в базі даних, якщо за цим номером автомобіля було знайдено користувача, то повертаємо його до фронтенд частини, інакше генеруємо помилку. Повна реалізація серверної частини наведена в Додатку Г.

ВИСНОВКИ

В результаті роботи над дипломною роботою було розроблено інформаційну систему «Автоматизована парковка» з використанням фронтенд фреймворку Angular та бекенд фреймворку NestJS. Метою роботи було створення легкої та функціональної системи для менеджменту паркування.

У рамках роботи були досліджені сервіси для розпізнавання тексту та номерних знаків, що дозволило інтегрувати ці технології у розроблену систему.

Крім цього, ретельно вивчалися та застосовувалися основні принципи розробки Angular та NestJS застосунків, з метою забезпечення оптимальної швидкості та надійності системи.

Однією з ключових функцій, реалізованих у цій системі, була система верифікації користувачів за номером автомобіля. Цей функціонал дозволяє ефективно та безпечно контролювати доступ до парковки, використовуючи у якості унікального ідентифікатору – номер автомобіля.

Основні зусилля було спрямовано на створення зручного та ефективного інструменту для управління паркуванням, що поєднує в собі новітні технології розпізнавання номерів автомобілів та сучасні підходи до розробки фронтенду та бекенду системи.

У підсумку, розроблена інформаційна система «Автоматизована парковка» на основі фреймворків Angular та NestJS відповідає вимогам та цілям проєкту. Її функціонал, що включає в себе сучасний сервіс для розпізнавання тексту AWS Rekognition, а також систему верифікації користувачів за номерами автомобілів, відповідає потребам управління паркуванням, пропонуючи зручний та надійний інструмент для контролю доступу до паркінгу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Amazon Rekognition. Getting started with Amazon Rekognition. URL: https://docs.aws.amazon.com/rekognition/latest/dg/getting-started.html (дата звернення: 04.09.2023).
2. Angular. Angular Developer Documentation. URL: https://angular.io/docs/ (дата звернення: 02.09.2023).
3. TypeScript. TypeScript Developer Documentation. URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html (дата звернення: 02.09.2023).
4. Emmit S. J. A. SPA Design and Architecture: Understanding single-page web applications, Shelter Island, New York : Manning Publications Co., 2015.
5. NestJS. NestJS Developer Documentation. URL: https://docs.nestjs.com (дата звернення: 02.09.2023).
6. Freeman A. Pro Angular: Build Powerful and Dynamic Web Apps. London : Apress, 2022. 905 p.
7. Chebbi L. Reactive Patterns with RxJS for Angular: A practical guide to managing your Angular application’s data reactively and efficiently using RxJS 7. Birmingham : Packt Publishing, 2022. 224 p.
8. Bampakos A. Angular Projects: Build modern web apps by exploring Angular 12 with 10 different projects and cutting-edge technologies. Birmingham : Packt Publishing, 2021. 344 p.
9. Fain Y., Moiseev A. Angular Development with TypeScript. New York : Manning Publications, 2019. 560 p.
10. AWS S3. Amazon S3 Documtation. URL: https://aws.amazon.com/s3/?nc1=h_ls (дата звернення: 02.09.2023).

ДОДАТОК А

Angular-компонент реєстрації

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { AppService } from 'src/app/core/api/app.service';

export interface LPN {
 name?: string;
 surname?: string;
 email?: string;
 LPN: string;
 model?: string;
 parkspace?: string;
 createdAt?: string;
 updatedAt?: string;
 status?: boolean;
}

@Component({
 selector: 'app-admin-page',
 templateUrl: './admin-page.component.html',
 styleUrls: ['./admin-page.component.scss']
})
export class AdminPageComponent implements OnInit {
 fg = new FormGroup({
  name: new FormControl('', Validators.required),
  surname: new FormControl('', Validators.required),
  LPN: new FormControl('', Validators.required),
```

```
 email: new FormControl('', Validators.required),
 model: new FormControl('', Validators.required),
 parkspace: new FormControl('', Validators.required),
})

errorText: string = '';

constructor(private appService: AppService) {

}
ngOnInit(): void {
}

CreateUserCheck() {
 if(this.fg.invalid) {
  this.errorText = "Please fill in all required fields.";
 }
 const bodyUser = this.fg.value as LPN;
 const finalLPN: string = bodyUser.LPN.replaceAll('A', 'A').replaceAll('B',
'B').replaceAll('C', 'C')
  .replaceAll('E', 'E').replaceAll('H', 'H').replaceAll('I', 'I').replaceAll('K', 'K')
  .replaceAll('M', 'M').replaceAll('O', 'O').replaceAll('P', 'P').replaceAll('T',
'T').replaceAll('X', 'X');
 this.appService.createUser({
  ...bodyUser,
  LPN: finalLPN
 }).subscribe(data => console.log(data))
}
}
```

ДОДАТОК Б

Angular-сервіс

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { environment } from 'src/environments/environment';
import { HttpClient } from '@angular/common/http';
import { LPN } from 'src/app/components/admin-page/admin-page.component';

@Injectable({
 providedIn: 'root'
})
export class AppService {
 Api_url: string = environment.API_URL


 constructor(private httpClient: HttpClient) { }

 getLink(file: File): Observable<{ url: string }> {
  return this.httpClient.post<{ url: string }>(
   `${this.Api_url}/upload/link`,
   { filename: file.name, size: file.size }
  );
 }

 uploadImageToS3(url: string, file: File | undefined) {
  if (!file) {
   return of();
  }
```

```
 return this.httpClient.put(url, file);
}

detectTextfromPicture(image: string): Observable<{texts: string[]}> {
 return this.httpClient.post<{ texts: string[] }>(
  `${this.Api_url}/detect`,
  { image : image }
 );
}

createUser(user: LPN): Observable<LPN> {
 return this.httpClient.post<LPN>(
  `${this.Api_url}/lpn/create`,
  user
 );
}
verifyLpn(LPN: string): Observable<LPN> {
 return this.httpClient.post<LPN>(
  `${this.Api_url}/lpn/verify`,
  {LPN: LPN}
 );
}

getAll() {
 return this.httpClient.get(this.Api_url+'/lpn')
 }
}
```

ДОДАТОК В

Angular-компонент завантаження зображення

```
import { Component } from '@angular/core';
import { VerifyDialogComponent } from '../verify-dialog/verify-dialog.component';
import { MatDialog } from '@angular/material/dialog';
import {VerifyanswerService} from '../../core/api/verifyanswer.service'
import {VerifyInterface} from '../../core/interface/verifyanswer.interface'
import {AppService} from '../../core/api/app.service'
import { LPN } from '../admin-page/admin-page.component';
import { Observable, catchError, forkJoin, of } from 'rxjs';
import { HttpErrorResponse } from '@angular/common/http';

@Component({
 selector: 'app-uploadpicture',
 templateUrl: './uploadpicture.component.html',
 styleUrls: ['./uploadpicture.component.scss']
})
export class UploadPictureComponent {
 base64Image: string = '';
 imgToSave: string = '';
 loading: boolean = false;
 textDetected: string[]= [];
 PlateNumberRegex: RegExp = new
RegExp(/^[AABBCCEEHHIIKKMMOOPPTTXX]{2}\d{4}(?<!0{4})[AABBCCEE
HHIIKKMMOOPPTTXX]{2}$/)

 VerifyResult: VerifyInterface[] = [];
```

```
 constructor(private matDialog:MatDialog, private VerifyanswerService:
VerifyanswerService, private appService: AppService) {
 }
 getVerify() {
  this.VerifyResult = this.VerifyanswerService.getVerifyResult();
  console.log(this.VerifyResult);
 }
 detectText(){
  this.appService.detectTextfromPicture(this.imgToSave).subscribe((data:{texts:
string[]}) => {
   this.textDetected = data.texts;
   this.processLpnResponse();
  })
 }
 getUniqueValues(res: string[]) {
  return [...new Set(res)]
 }
 processLpnResponse() {
  let res = this.textDetected.filter(text => this.PlateNumberRegex.test(text.replaceAll(/
/g, ''))).map(text => {
   return text.replaceAll(/ /g, '').replaceAll('A', 'A').replaceAll('B', 'B').replaceAll('C',
'C')
   .replaceAll('E', 'E').replaceAll('H', 'H').replaceAll('I', 'I').replaceAll('K', 'K')
   .replaceAll('M', 'M').replaceAll('O', 'O').replaceAll('P', 'P').replaceAll('T',
'T').replaceAll('X', 'X')
  });
  if(!res.length) {
   console.log('LPN is not detected');
   return;
  }
```

```
  const LPNS = this.getUniqueValues(res);
  this.componeVerifyLpnRequests(LPNS);
 }
 componeVerifyLpnRequests(lpnsToVerify: string[]) {
  const requests: Observable<LPN | null>[] = lpnsToVerify.map((lpn) =>
this.verifyLPN(lpn));
  const reqs = forkJoin([...requests]).pipe(
   catchError((err: HttpErrorResponse) => {
    console.log( err.error.message || err.message );
    this.loading = false;
    return of();
   })
  );
  reqs.subscribe((data: (LPN | null)[]) => {
   console.log(data);
   const success: (LPN| null)[] = data.filter(item => Boolean(item));
   if(success.length) {
    this.openDialog({...success[0]!, status: true})
   }else {
    this.openDialog({LPN: lpnsToVerify[0], status: false})
   }
  })
 }

 verifyLPN(lpn: string): Observable<LPN | null> {
  this.loading = true;
  console.log(lpn);
  return this.appService.verifyLpn(lpn).pipe(
   catchError((err: HttpErrorResponse) => {
    console.log( err.error.message || err.message );
```

```
   this.loading = false;
   return of(null);
  })
 )
}

openDialog(lpn: LPN) {
 this.matDialog.open(VerifyDialogComponent,{
  width: '350px',
  height: '450px',
  panelClass: "modal-class",
  enterAnimationDuration: 0,
  exitAnimationDuration: 0,
  data: lpn
 })
}
file: File | undefined = undefined ;

onFileChange(event: any) {
 this.file = event.target.files[0];
 if(!this.file) return
 this.convertToBase64();
 this.appService.getLink(this.file).subscribe((data: {url: string}) => {
  if (!data || !this.file) return;
  this.uploadImg(data.url, this.file);
 })
}

convertToBase64() {
```

```
  if (!this.file) {
   return;
  }

  const reader = new FileReader();
  reader.onload = () => {
   this.base64Image = reader.result as string;
  };
  reader.readAsDataURL(this.file);



 }

uploadImg(url: string, file: File) {
  const formattedUrl: URL = new URL(url);
  this.imgToSave = formattedUrl.pathname.replace('/', '');
  this.appService.uploadImageToS3(url, file).subscribe((data: any) => {
   this.loading = false;
  });
 };
}
```

ДОДАТОК Г

Серверна частина застосунку

Г.1 Upload Service

```
import { PutObjectCommand, S3Client } from '@aws-sdk/client-s3';
import { getSignedUrl } from '@aws-sdk/s3-request-presigner';
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class UploadService {

 private s3Client = new S3Client({
  region: this.configService.getOrThrow('AWS_S3_REGION'),
 });

 constructor(private configService: ConfigService) {}

 async uploadFileToS3(file: Express.Multer.File) {
  const bucket: string = this.configService.getOrThrow('AWS_BUCKET_NAME');
  return this.s3Client.send(
    new PutObjectCommand({
      Bucket: bucket,
      Key: file.fieldname,
      Body: file.buffer,
    })
  )
 }
```

```
 async getFileUrl(file: {filename: string; size: number}) {
  const bucket: string = this.configService.getOrThrow('AWS_BUCKET_NAME');
  const command = new PutObjectCommand({
     Bucket: bucket,
     Key: file.filename,
     ContentLength: file.size
    });
  const url = await getSignedUrl(this.s3Client, command, { expiresIn: 1200 });
  return {url: url};
 }
}
```

Γ.2 Detect Service

```
import { DetectTextCommand, DetectTextCommandInput, DetectTextCommandOutput, RekognitionClient, TextDetection } from '@aws-sdk/client-rekognition';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class DetectService {

  private client = new RekognitionClient({
    region: this.configService.getOrThrow('AWS_S3_REGION')
  });
  private readonly bucket: string = this.configService.getOrThrow('AWS_BUCKET_NAME');
```

```
constructor(
  private readonly configService: ConfigService,
) {}

async detectText(
  body: { image: string }
) {
  const input: DetectTextCommandInput = {
    Filters: {
      WordFilter: {
        MinConfidence: 10,
      }
    },
    Image: {
      S3Object: {
        Bucket: this.bucket,
        Name: body.image,
      },
    },
  };
  const command = new DetectTextCommand(input);
  let response: DetectTextCommandOutput | undefined = undefined;
  try {
    response = await this.client.send(command);
  } catch (err) {
    console.log(err);

    throw new HttpException(
      'Error: Something went wrong while trying to detect texts!',
      HttpStatus.BAD_REQUEST,
```

```
    );
  }
  console.log(response);
  const texts: string[] = response ? response.TextDetections.map((detection: TextDetection) => detection.DetectedText) : [];
  return {texts: texts };
 }
}
```

Γ.3 Lpn Service

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { Lpn, LpnDocument } from './schemas/lpn.schema';
import { CreateLpnDto } from './dto/lpn.dto';

@Injectable()
export class LpnService {

 constructor(
  @InjectModel(Lpn.name) private lpnModel: Model<LpnDocument>,
 ) {}

 async createLpn(body: CreateLpnDto) {
  const lpnEntity = await this.lpnModel.findOne({
   LPN: body.LPN,
  });
  if (lpnEntity) {
```

```
      throw new UnauthorizedException('LPN is already exists');
    }
    const newLpn = new this.lpnModel(body);
    const createdLpn = await newLpn.save();
    console.log("Created User: ",createdLpn);
    return createdLpn;
  }

  async getLpn(body: {LPN: string}) {
    const lpn = await this.lpnModel.findOne({ LPN: body.LPN });
    if (!lpn) {
      throw new UnauthorizedException('LPN does not exist');
    }
    return lpn;
  }
}
```