

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Масштабований сервіс для генерації скріншотів»

Виконав студент групи ІПЗ-22-4

ПОГОЙДАШ Іван Сергійович

Керівник роботи:

ЄФРЕМОВ Юрій Миколайович

Рецензент:

САВІЦЬКИЙ Роман Святославович

Житомир – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна Вакалюк
«27» лютого 2026 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ПОГОЙДАШ Іван Сергійович**

Керівник роботи: **ЄФРЕМОВ Юрій Миколайович**

Тема роботи: **Масштабований сервіс для генерації скріншотів**

затверджена Наказом закладу вищої освіти від **«27» лютого 2026 р., №92/с**

Вихідні дані для роботи: **процес автоматизованої генерації скріншотів вебсторінок у розподілених вебсистемах.**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Єфремов Ю.М.	27.02.2026р.	28.02.2026р.
2	Єфремов Ю.М.	27.02.2026р.	28.02.2026р.
3	Єфремов Ю.М.	27.02.2026р.	28.02.2026р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з вебзастосунку сервісу та пояснювальної записки. Пояснювальна записка до випускної роботи містить текстову частину, викладену на 85 сторінках друкованого тексту. Список використаних джерел містить 30 найменувань і займає 3 сторінки. У роботі наведено 52 рисунків та 40 таблиць. Також у роботі є 27 сторінок додатків.

Метою роботи є розробка масштабованого сервісу для генерації скріншотів вебсторінок із використанням мікросервісної архітектури та сучасних хмарних технологій.

У першому розділі поставлено задачу кваліфікаційної роботи, визначено ключовий функціонал сервісу генерації скріншотів, проведено аналіз існуючих аналогів, а також обґрунтовано вибір архітектури та технологій розробки.

У другому розділі визначено вимоги до системи, змодельовано варіанти використання та архітектуру сервісу, розроблено структуру бази даних, спроектовано алгоритми обробки запитів і генерації скріншотів, а також реалізовано функціональність серверної та клієнтської частин із наведенням ключових фрагментів програмного коду.

У третьому розділі описано процес розгортання мікросервісної системи, користувацький інтерфейс застосунку, а також проведено тестування розробленого сервісу.

У результаті виконання роботи розроблено програмний сервіс у вигляді вебзастосунку та API, що забезпечує автоматизовану генерацію скріншотів вебсторінок із підтримкою масштабування, черг обробки запитів, інтеграції з клієнтськими системами та гнучкого налаштування параметрів захоплення зображень.

					ІПЗ.КР.Б-121-26-ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Погойдаш І.С.			Масштабований сервіс для генерації скріншотів Пояснювальна записка	Літ.	Арк.	Аркушів
Керівник		Єфремов Ю.М.					3	115
Н. контр.		Тичина О.П.				Житомирська політехніка, група ІПЗ-22-4		
Зав. каф.		Вакалюк Т.А.						

КЛЮЧОВІ СЛОВА: ГЕНЕРАЦІЯ СКРІНШОТІВ, ВЕБСЕРВІС, МІКРОСЕРВІСНА АРХІТЕКТУРА, RABBITMQ, PLAYWRIGHT, ХМАРНІ ТЕХНОЛОГІЇ, МАСШТАБОВАНІСТЬ.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ABSTRACT

The bachelor's qualification work consists of a web-based service application and an explanatory note. The explanatory note contains the main text presented on 85 pages of printed text. The list of references includes 30 sources and occupies 3 pages. The work contains 52 figures and 40 tables. Additionally, there are 27 pages of appendices.

The aim of this work is to develop a scalable service for generating website screenshots using microservices architecture and modern cloud technologies.

The first chapter defines the objectives of the qualification work, identifies the key functionality of the screenshot generation service, analyzes existing solutions, and justifies the choice of architecture and development technologies.

The second chapter describes system requirements, models use cases and system architecture, designs the database structure, develops algorithms for request processing and screenshot generation, and implements both backend and frontend functionality with key code examples.

The third chapter describes the deployment process of the microservices-based system, the user interface of the application, and the testing of the developed service.

As a result, a software solution in the form of a web application and API has been developed, providing automated website screenshot generation with support for scalability, request queuing, integration with external systems, and flexible configuration of capture parameters.

KEYWORDS: SCREENSHOT GENERATION, WEB SERVICE, MICROSERVICES ARCHITECTURE, RABBITMQ, PLAYWRIGHT, CLOUD TECHNOLOGIES, SCALABILITY.

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		5

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП	8
РОЗДІЛ 1 АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ СИСТЕМИ	10
1.1 Постановка задачі.....	10
1.2 Аналіз аналогів програмного продукту	12
1.3 Вибір архітектури системи.....	20
1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	26
Висновки до першого розділу.....	29
РОЗДІЛ 2 ПРОЄКТУВАННЯ СИСТЕМИ.....	30
2.1 Визначення варіантів використання та архітектура сервісу.....	30
2.2 Розробка бази даних.....	45
2.3 Проєктування та реалізація алгоритмів системи	55
2.4 Реалізація функціоналу системи генерації скріншотів вебсторінок.....	58
Висновки до другого розділу	62
РОЗДІЛ 3 ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ ІЗ СИСТЕМОЮ	63
3.1 Порядок встановлення та налаштування параметрів системи	63
3.2 Структура інтерфейсу платформи.....	66
3.3 Тестування програмного системи	82
Висновки до третього розділу.....	84
ВИСНОВКИ.....	85
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	86
ДОДАТКИ.....	89

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API - Application Programming Interface

СУБД - Система управління базами даних

AMQP - Advanced Message Queuing Protocol

UML - Unified Modeling Language

ORM - Object-Relational Mapping

SQL - Structured Query Language

ASP.NET - Active Server Pages .NET

SaaS - Software as a Service

JSX - JavaScript Syntax Extension

SSR - Server-Side Rendering

HTTP - HyperText Transfer Protocol

REST - Representational State Transfer

RabbitMQ - Брокер повідомлень для асинхронної комунікації

JWT - JSON Web Token

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		7

ВСТУП

Актуальність теми. Сучасний світ переживає стрімкі технологічні зміни, що суттєво впливають на розвиток інформаційних систем і сервісів, зокрема вебзастосунків. Одним із актуальних напрямків є створення масштабованих хмарних сервісів, які дозволяють автоматизувати рутинні процеси обробки та генерації даних у вебсередовищі. У цьому контексті особливого значення набувають сервіси генерації скріншотів вебсторінок, які використовуються в тестуванні програмного забезпечення (QA), моніторингу вебресурсів, маркетинговій та SEO-аналітиці, кібербезпеці, legal tech для фіксації вебконтенту, медіа-архівзації та DevOps-процесах. Це підтверджує їхню практичну значущість і широке використання в сучасному бізнес-середовищі. Сервіс генерації скріншотів вебсторінок - це програмна система, що дозволяє користувачам або зовнішнім системам автоматично отримувати зображення вебсторінок із можливістю гнучкого налаштування параметрів захоплення контенту. Такі сервіси широко застосовуються для end-to-end тестування, візуальної валідації інтерфейсів, архівування вебконтенту та інтеграції у CI/CD процеси. В умовах зростання кількості вебзастосунків та складності їх інтерфейсів потреба у подібних інструментах постійно збільшується, що підкреслює актуальність розробки масштабованого та продуктивного рішення.

Метою випускної кваліфікаційної роботи бакалавра є розробка масштабованого вебсервісу для генерації скріншотів вебсторінок із використанням сучасних технологій та мікросервісної архітектури. Для досягнення поставленої мети в роботі необхідно вирішити низку завдань, що охоплюють аналіз існуючих аналогів сервісів генерації скріншотів з визначенням їхніх переваг і недоліків, розробку архітектури системи та визначення її основних компонентів, а також вибір відповідних інструментів і технологій для реалізації сервісу. Крім того, передбачається реалізація основного функціоналу системи та проведення тестування розробленого програмного рішення.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		8

Об’єктом дослідження є процес автоматизованої генерації скріншотів вебсторінок у розподілених вебсистемах.

Предметом дослідження є застосування сучасних вебтехнологій, мікросервісної архітектури та черг повідомлень для побудови масштабованого сервісу генерації скріншотів.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		9

РОЗДІЛ 1 АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ СИСТЕМИ

1.1 Постановка задачі

Призначення системи полягає у розробці масштабованого вебсервісу для автоматизованої генерації скріншотів вебсторінок, який може використовуватись як кінцевими користувачами через вебінтерфейс, так і зовнішніми системами через REST API. Сервіс забезпечує можливість отримання зображень вебсторінок із гнучким налаштуванням параметрів рендерингу, що дозволяє адаптувати процес генерації під різні сценарії використання, включаючи тестування вебзастосунків, моніторинг контенту, автоматизацію перевірок інтерфейсів та інтеграцію в CI/CD процеси.

Метою впровадження системи є створення високопродуктивного та масштабованого рішення, здатного обробляти велику кількість паралельних запитів на генерацію скріншотів із мінімальними затримками. Система повинна забезпечувати асинхронну обробку задач, розподіл навантаження між worker-сервісами, а також пріоритизацію запитів залежно від тарифного плану користувача. Додатково передбачається підтримка API-токенів для інтеграції сторонніх систем, що дозволяє використовувати сервіс у зовнішніх застосунках та автоматизованих сценаріях.

Окремою важливою складовою системи є механізм зберігання згенерованих скріншотів. Після обробки запиту результуючі зображення зберігаються у зовнішньому об'єктному сховищі (blob storage), а в базі даних зберігаються лише метадані та посилання на файл. Такий підхід дозволяє зменшити навантаження на основну базу даних та забезпечити ефективне масштабування системи зберігання.

Також система містить механізм тарифікації, заснований на використанні внутрішніх "points" (абстрактних облікових одиниць). Кожна операція генерації скріншота списує певну кількість балів залежно від складності запиту. Вартість розраховується динамічно з урахуванням

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		10

параметрів запиту, таких як розмір сторінки, тип захоплення, додаткові опції рендерингу та рівень складності обробки сторінки.

У межах аналізу функціональних вимог передбачається дослідження потреб різних категорій користувачів системи, серед яких індивідуальні розробники, бізнес-клієнти, а також команди тестування, що використовують інструменти автоматизації UI та E2E перевірок. На основі цього визначаються основні функції системи, зокрема генерація скріншотів за URL-адресою, підтримка різних форматів зображень (PNG, JPEG), можливість вибору типу захоплення (повна сторінка, окремий елемент, модальне вікно або виділена область), а також розширені параметри рендерингу. Окремо формуються нефункціональні вимоги, що охоплюють масштабованість, відмовостійкість, безпеку API, продуктивність, контроль навантаження та стабільність роботи headless-браузерів.

Реалізація системи базується на мікросервісному підході з асинхронною взаємодією компонентів через брокер повідомлень. Як основний технологічний стек обрано ASP.NET Core для реалізації API Gateway та серверних сервісів, React 18 для клієнтської частини, PostgreSQL як основну реляційну базу даних, RabbitMQ для організації черг завдань і забезпечення асинхронної обробки, а також Playwright як headless-браузер для генерації скріншотів. Архітектура системи передбачає використання низки патернів проєктування, зокрема Inversion of Control, реалізованого через dependency injection в ASP.NET Core, Repository Pattern для абстракції доступу до даних, а також Result Pattern для уніфікованої обробки результатів операцій. Загальна структура побудована за принципом мікросервісів із внутрішнім модульним поділом (Feature Modules та Shared Kernel), що дозволяє зменшити складність бізнес-логіки без надмірного ускладнення архітектури.

Для забезпечення ефективної взаємодії між API Gateway та worker-сервісами додатково використовується gRPC, що дозволяє досягти низької затримки та забезпечує швидку синхронну взаємодію між внутрішніми

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		11

компонентами системи. У той же час RabbitMQ застосовується як механізм асинхронного чергування задач генерації скріншотів, тоді як gRPC використовується для обміну службовими даними, зокрема отримання статусів виконання задач, передачі результатів та метаданих.

Згенеровані скріншоти зберігаються у зовнішньому blob-сховищі, що дозволяє ефективно масштабувати систему зберігання файлів. У реляційній базі даних зберігаються дані користувачів, метадані скріншотів, статуси виконання задач, а також посилання на файли у сховищі. Такий підхід дає змогу зменшити обсяг даних у базі та підвищити загальну продуктивність системи.

Серверна частина реалізується на базі ASP.NET Core із використанням DI-контейнера та модульної структури застосунку. Інтеграція з RabbitMQ забезпечує асинхронну обробку задач, а також реалізацію черг із пріоритетами відповідно до тарифних планів користувачів (Regular, Pro, Advanced). Клієнтська частина реалізується на основі React 18 із використанням Redux для управління станом застосунку. Вона забезпечує функціонал створення запитів на генерацію скріншотів, перегляд історії запитів та моніторинг статусів виконання задач.

У процесі розробки передбачається виконання модульного та інтеграційного тестування ключових компонентів системи. Окрему увагу приділено тестуванню взаємодії між чергами повідомлень, API Gateway та worker-сервісами. Оптимізація продуктивності включає повторне використання browser context у Playwright, зменшення накладних витрат на створення процесів, а також контроль паралельного виконання задач. Додатково впроваджуються механізми retry, rate limiting і захист від перевантаження системи, що забезпечує стабільність її роботи під навантаженням.

1.2 Аналіз аналогів програмного продукту

На сучасному ринку програмних рішень для автоматизованої генерації скріншотів вебсторінок існує значна кількість комерційних та open-source

сервісів, які надають доступ до функціональності захоплення вебконтенту через API або інструменти автоматизації браузера. Такі системи використовуються в різних доменах, включаючи тестування вебзастосунків, моніторинг інтерфейсів, генерацію прев'ю сторінок, архівацію вебконтенту та інтеграцію у CI/CD процеси.

З метою розробки конкурентоспроможного програмного продукту було проведено аналіз найбільш поширених рішень, а саме: Urlbox, ScreenshotAPI, Browserless. Додатково розглянуто загальний клас self-hosted рішень на основі Playwright, оскільки Playwright часто використовуються як альтернатива SaaS-платформам.

Порівняльний аналіз зазначених рішень здійснюється за низкою ключових критеріїв, що охоплюють функціональні можливості генерації скріншотів, рівень кастомізації процесу рендерингу сторінки, підтримку складних UI-сценаріїв, можливості інтеграції через API, масштабованість і продуктивність систем, модель тарифікації, підтримку асинхронної обробки запитів, механізми зберігання результатів, а також підтримку розширених параметрів запиту, таких як headers, cookies, locale та emulation пристроїв. Додатково враховується можливість контролю складності виконання запитів і поведінки рендерингу.

Urlbox (Рис. 1.1) є одним із найбільш відомих enterprise-рішень для генерації скріншотів вебсторінок, PDF-документів та інших візуальних представлень вебконтенту. Сервіс орієнтований на бізнес-клієнтів та надає високорівневий API для інтеграції у сторонні системи.

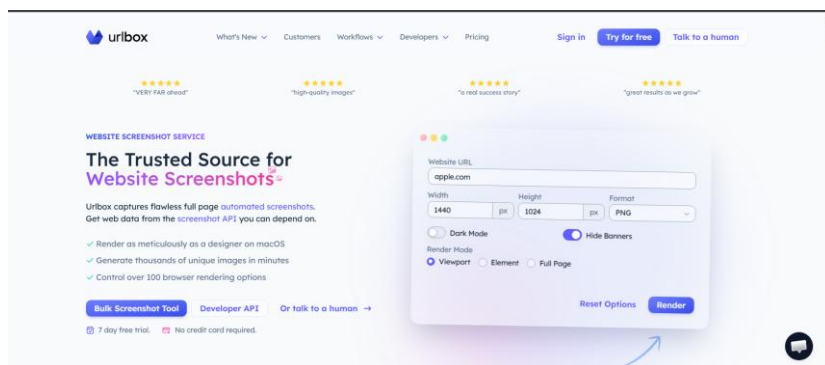


Рис. 1.1. - Вигляд вебсервісу для генерації скріншотів Urlbox

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				13
Змн.	Арк.	№ докум.	Підпис	Дата		

Urlbox підтримує широкий набір функціональних можливостей, що охоплюють генерацію скріншотів повної сторінки (full-page rendering), захоплення viewport або окремої області сторінки, а також експорт результатів у різних форматах, зокрема PNG, JPEG, WEBP та PDF. Додатково сервіс забезпечує емуляцію різних типів пристроїв (mobile, tablet, desktop), налаштування розміру viewport, роботу з cookies та HTTP headers, а також можливість блокування окремих типів ресурсів сторінки, таких як рекламні елементи, скрипти або зображення. Також реалізовано базову підтримку очікування повного завантаження сторінки, зокрема через механізм network idle.

З архітектурної точки зору Urlbox реалізовано як хмарний SaaS-сервіс із централізованою інфраструктурою виконання запитів. Взаємодія з системою здійснюється через REST API, тоді як внутрішня реалізація процесів рендерингу та обробки запитів прихована від кінцевого користувача.

Серед основних переваг Urlbox можна виділити стабільність роботи, що забезпечується enterprise-рівнем хмарної інфраструктури, високу якість рендерингу вебсторінок, підтримку широкого спектра форматів експорту, відносно простий інтерфейс API для інтеграції, а також наявність базових сценаріїв автоматизації. Водночас сервіс має і низку недоліків. Зокрема, до них належать висока вартість використання при збільшенні кількості запитів, відсутність гнучкої моделі тарифікації залежно від складності виконуваних операцій, обмежені можливості кастомізації запитів, а також відсутність доступу до внутрішнього execution pipeline. Крім того, сервіс має обмежену підтримку складних сценаріїв взаємодії з DOM, таких як багатокрокові interaction flows.

ScreenshotAPI (Рис 1.2) є більш простим сервісом, який орієнтований на швидку генерацію скріншотів вебсторінок через HTTP API. Його основна мета - надати мінімалістичний інтерфейс для отримання зображення сторінки без складної конфігурації.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		14

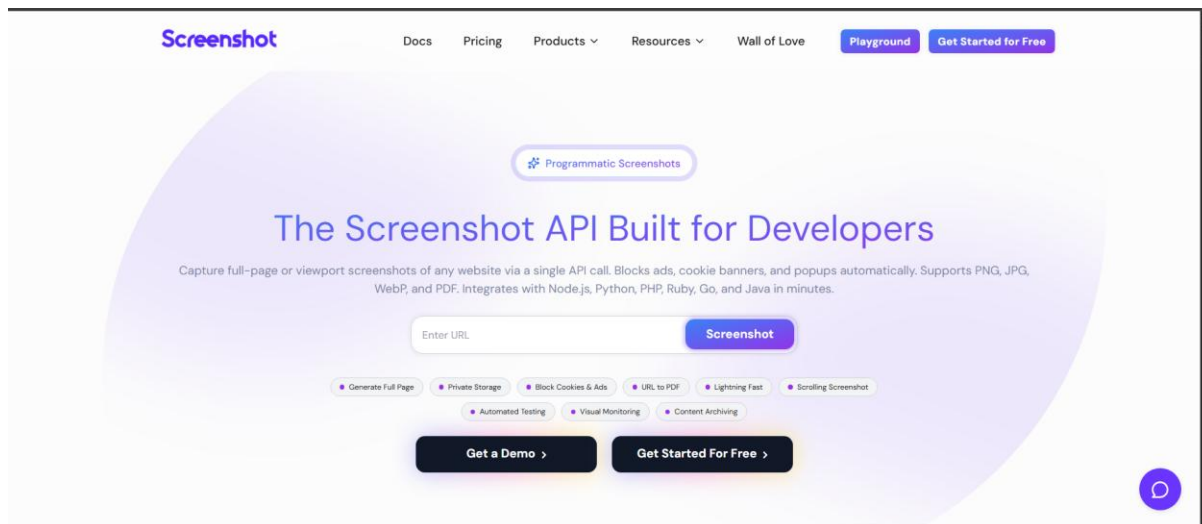


Рис. 1.2. - Вигляд вебсервісу для генерації скріншотів ScreenshotAPI

ScreenshotAPI надає обмежений, але достатній для базових сценаріїв набір функціональних можливостей. Зокрема, сервіс забезпечує отримання скріншота вебсторінки за URL-адресою, підтримку основних форматів зображень (PNG, JPEG), а також можливість вибору режиму захоплення сторінки - full-page або viewport. Додатково користувач може задавати розмір viewport, а також використовувати базові параметри таймауту виконання запиту. Загалом сервіс пропонує простий REST API без складних налаштувань або розширеної конфігурації.

З архітектурної точки зору ScreenshotAPI реалізовано як легкий API gateway до внутрішнього headless browser engine. Основний акцент у проєктуванні зроблено на простоті інтеграції та мінімізації складності використання, а не на гнучкості або підтримці розширених сценаріїв обробки.

Серед ключових переваг сервісу слід відзначити низький поріг входу для розробників, мінімальну необхідність у конфігурації, швидке отримання результатів виконання запитів, а також придатність для реалізації простих use-case сценаріїв. Водночас сервіс має низку суттєвих обмежень. До них належать відсутність підтримки складних сценаріїв взаємодії з інтерфейсом (наприклад, element capture, модальні вікна або динамічні стани UI), обмежені можливості кастомізації процесу рендерингу, відсутність

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				15
Змн.	Арк.	№ докум.	Підпис	Дата		

розширених параметрів запиту (cookies, headers, locale), а також відсутність складної системи чергування або пріоритетизації задач. Крім того, сервіс не передбачає гнучкої тарифікації, має обмежену масштабованість при високому навантаженні та не надає прозорості щодо внутрішньої архітектури.

Browserless (Рис 1.3) представляє інший клас рішень - це платформа, що надає віддалений доступ до headless-браузерів (Chromium) через API (Puppeteer/Playwright-like environment). Це більш низькорівневе рішення, яке дозволяє виконувати довільні сценарії автоматизації браузера.

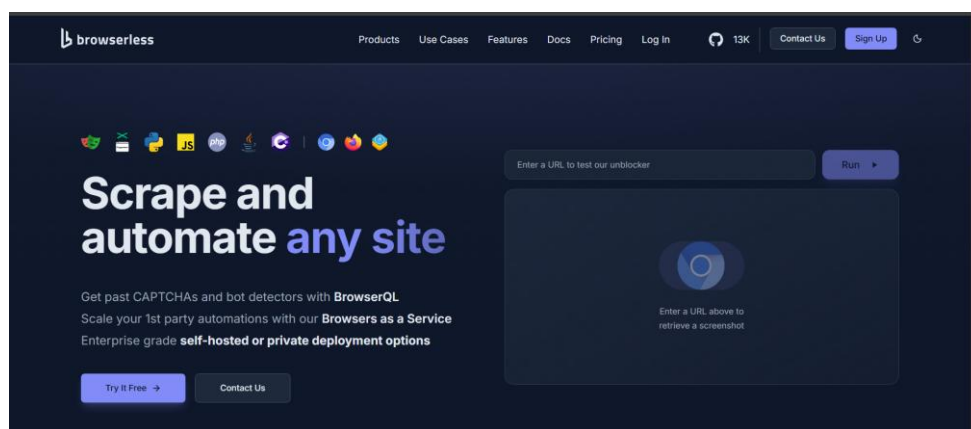


Рис. 1.3. - Вигляд вебсервісу для генерації скріншотів Browserless

Browserless надає широкий набір функціональних можливостей, пов'язаних із виконанням браузерної автоматизації у хмарному середовищі. Зокрема, сервіс забезпечує запуск headless браузерів, виконання скриптів на базі Puppeteer або Playwright, генерацію скріншотів та PDF-документів, а також виконання JavaScript-коду безпосередньо в контексті вебсторінки. Додатково підтримуються WebSocket-з'єднання для довготривалих сесій, що дозволяє реалізовувати складні сценарії взаємодії. Сервіс також забезпечує повний контроль DOM-структури та підтримує складні багатокрокові сценарії автоматизації.

З архітектурної точки зору Browserless не є високорівневим API для генерації скріншотів, а радше інфраструктурним сервісом, що надає runtime для виконання браузерних сесій. Уся логіка побудови сценаріїв, управління

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		16

процесом рендерингу та обробки результатів покладається безпосередньо на розробника.

Серед основних переваг сервісу варто відзначити максимальну гнучкість і повний контроль над процесом автоматизації, можливість реалізації складних багатокрокових сценаріїв взаємодії, повну сумісність із Playwright, а також придатність для інтеграції у складні системи автоматизації. Водночас Browserless має низку обмежень. Зокрема, відсутня високорівнева бізнес-логіка, що ускладнює використання для простих задач. Використання сервісу потребує написання власних скриптів для кожного сценарію, що підвищує складність розробки. Крім того, масштабування великої кількості користувачьких сценаріїв може вимагати додаткових зусиль, оскільки відсутні вбудовані механізми бізнес-рівня, такі як тарифікація, контроль складності запитів або система чергування задач. Це, у свою чергу, передбачає необхідність реалізації власної системи оркестрації та керування навантаженням.

Окремо слід виділити self-hosted підхід, коли розробники використовують Playwright (Рис 1.4) для побудови власного сервісу генерації скріншотів.

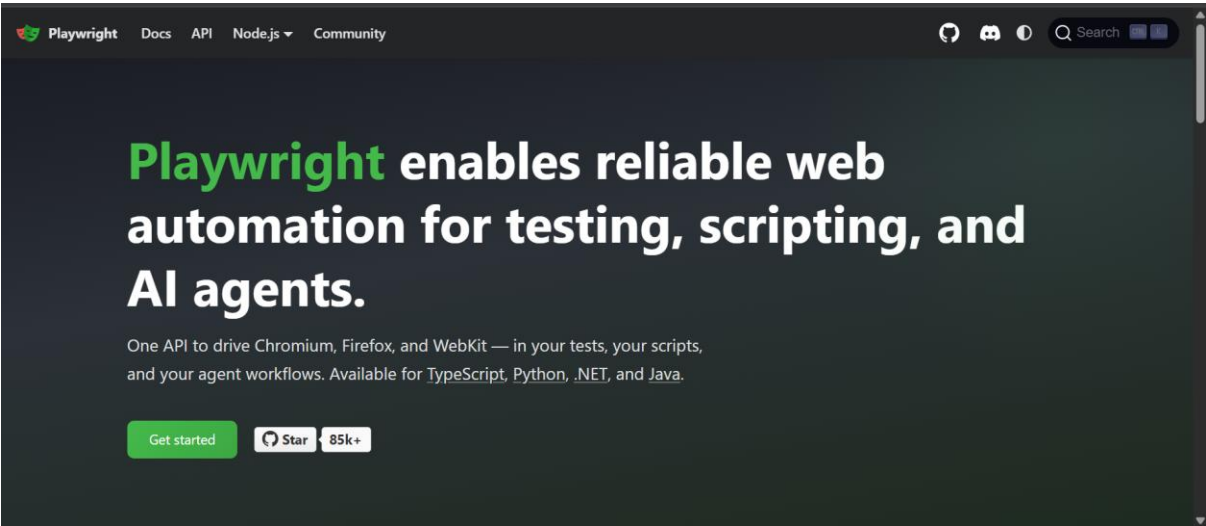


Рис. 1.4. - Вигляд документації open-source рішення Playwright для роботи з headless браузерми

Self-hosted рішення на основі Playwright надають повний контроль над браузерним середовищем та процесом рендерингу вебсторінок. Такі підходи

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		17

дозволяють реалізовувати будь-яку логіку взаємодії з вебінтерфейсом, підтримують створення кастомних сценаріїв автоматизації та не обмежуються рамками SaaS-платформ.

Додатковою перевагою є можливість глибокої інтеграції з власною інфраструктурою та внутрішніми сервісами організації. Серед ключових переваг такого підходу слід відзначити повну кастомізацію системи під конкретні бізнес-вимоги, відсутність vendor lock-in, відносно низьку вартість використання (за винятком витрат на інфраструктуру), а також можливість реалізації будь-якої необхідної бізнес-логіки без обмежень з боку сторонніх API. Водночас self-hosted підхід має і суттєві недоліки.

Зокрема, він вимагає самостійного забезпечення масштабування системи, побудови fault-tolerant архітектури, а також реалізації таких механізмів, як черги задач, retry-логіка та rate limiting. Додатково необхідно враховувати значні витрати на підтримку інфраструктури та відсутність готової API-моделі, що ускладнює швидкий старт розробки та підвищує загальну складність впровадження рішення.

Порівняльний аналіз існуючих рішень

Нижче наведено порівняльну таблицю основних систем за ключовими характеристиками у контексті функціональності та архітектури.

Таблиця 1.1

Характеристика	Urlbox	Screenshot API	Browserless	Self-hosted Playwright	Запропонована система
REST API для скріншотів	+	+	+	±	+
Full-page screenshot	+	+	+	+	+
Скріншот елемента (selector-based)	±	-	+	+	+

Підтримка модальних вікон / UI станів	±	-	+	+	+
Динамічний рендеринг (JS-heavy pages)	+	±	+	+	+
Підтримка headers / cookies / locale	+	-	+	+	+
Блокування ресурсів сторінки	+	-	±	+	+
Гнучка тарифікація (points-based model)	-	-	-	-	+
API токени з градацією доступів (read/write)	-	-	-	-	+
Контроль складності запиту	-	-	-	-	+
Інтеграція у CI/CD pipeline	+	+	+	+	+

Проведений аналіз показує, що існуючі рішення умовно можна поділити на три основні категорії.

До першої категорії належать SaaS API-сервіси (зокрема Urlbox та ScreenshotAPI), які забезпечують простий і швидкий доступ до функціональності генерації скріншотів. Водночас такі рішення характеризуються обмеженою гнучкістю налаштувань та недостатньою бізнес-кастомізацією.

Другу категорію становлять infrastructure-level сервіси, такі як Browserless, які надають повний контроль над браузерним середовищем і процесом виконання сценаріїв. Однак їх використання потребує самостійної реалізації всієї бізнес-логіки, включно з оркестрацією задач, обробкою результатів та управлінням навантаженням.

Третя категорія включає self-hosted рішення на основі інструментів автоматизації браузера, таких як Playwright або Puppeteer. Вони забезпечують максимальну гнучкість і контроль над виконанням сценаріїв, проте вимагають значних інженерних зусиль для побудови повноцінної масштабованої системи, включаючи інфраструктуру черг, балансування навантаження та забезпечення відмовостійкості.

Запропонована у роботі система поєднує переваги всіх зазначених підходів. Зокрема, вона надає високорівневий API, подібний до SaaS-рішень, забезпечує гнучкість автоматизації на рівні Browserless, а також зберігає контроль над інфраструктурою, характерний для self-hosted систем. Додатково реалізовано унікальну модель тарифікації на основі складності запиту (points model), що дозволяє більш точно оцінювати навантаження та ефективніше розподіляти ресурси, і яка відсутня в існуючих аналогах.

1.3 Вибір архітектури системи

Вибір архітектури для системи генерації скріншотів вебсторінок є одним із ключових етапів проєктування, оскільки він визначає подальшу масштабованість, продуктивність, відмовостійкість та гнучкість системи [7], [9]. Оскільки запропоноване рішення є SaaS-платформою з високим навантаженням та асинхронною обробкою запитів, архітектурні рішення повинні забезпечувати ефективну обробку великої кількості конкурентних задач [7], а також можливість горизонтального масштабування окремих компонентів.

Архітектура системи впливає на спосіб взаємодії між компонентами, рівень зв'язності між модулями, підхід до масштабування та управління станом системи. У контексті даної роботи розглядаються три основні архітектурні підходи: монолітна архітектура, мікросервісна архітектура та сервісно-орієнтована архітектура (SOA).

Монолітна архітектура (Рис 1.5) передбачає побудову системи як єдиного застосунку, у якому всі компоненти (API, бізнес-логіка, доступ до даних, обробка запитів) тісно інтегровані та розгортаються як один артефакт.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		20

У такому підході всі модулі використовують спільний процес виконання та зазвичай одну базу даних [12].

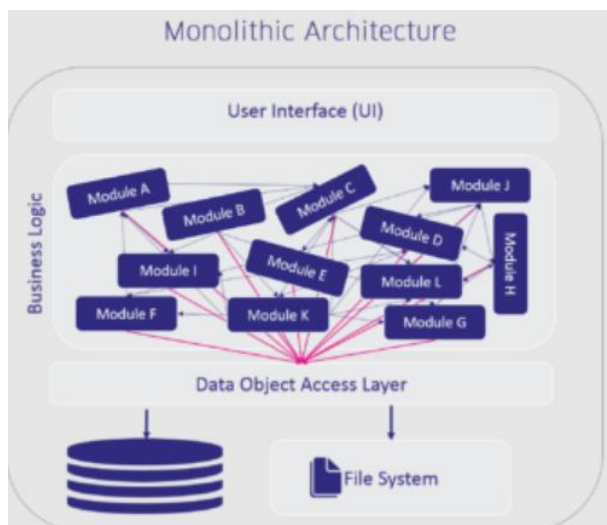


Рис 1.5. - Візуальне зображення монолітної архітектури

Монолітна архітектура характеризується єдиною кодовою базою, спільним процесом розгортання всіх компонентів, централізованою базою даних, а також синхронною взаємодією між модулями всередині одного процесу.

Серед основних переваг такого підходу можна виділити простоту початкової розробки, легкість розгортання на ранніх етапах проєкту, відсутність складної міжсервісної комунікації, а також відносно просту налагоджуваність у межах невеликих систем. Водночас монолітна архітектура має низку суттєвих недоліків. Зокрема, ускладнюється масштабування окремих компонентів системи (наприклад, worker-частини), виникає ризик появи «вузьких місць» при зростанні навантаження, а також підвищується складність підтримки коду в міру збільшення розміру проєкту. Крім того, такий підхід не дозволяє здійснювати незалежне масштабування окремих частин системи, наприклад API та обробки скріншотів, а також характеризується зниженою відмовостійкістю, оскільки збій одного компонента може призвести до зупинки всієї системи.

З огляду на вимоги даного проєкту, монолітна архітектура є недоцільною, оскільки система повинна підтримувати високу паралельність

виконання задач та можливість горизонтального масштабування worker-компонентів.

Сервісно-орієнтована архітектура (SOA) (Рис 1.6) передбачає побудову системи у вигляді набору сервісів, які взаємодіють між собою через стандартизовані протоколи (часто HTTP або SOAP) [10]. Кожен сервіс виконує певну бізнес-функцію та може використовуватись повторно в різних контекстах.

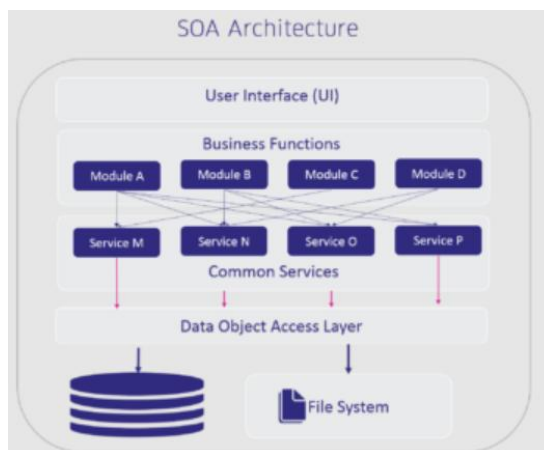


Рис 1.6. - Візуальне зображення SOA архітектури

SOA (Service-Oriented Architecture) передбачає побудову системи як набору незалежних сервісів, що взаємодіють між собою через стандартизовані протоколи обміну даними. Для даного підходу характерне використання централізованих або напівцентралізованих інтеграційних механізмів, а також орієнтація на повторне використання сервісів у межах різних бізнес-процесів.

До основних переваг SOA належать можливість повторного використання сервісів, гнучкість інтеграції із зовнішніми системами, модульність архітектури, а також підтримка гетерогенних технологічних стеків, що дозволяє поєднувати різноманітні компоненти в межах однієї системи. Водночас SOA має низку суттєвих недоліків. Зокрема, вона характеризується складністю конфігурації та супроводу, значними накладними витратами на інтеграційний шар, а також менш ефективною масштабованістю у порівнянні з сучасними мікросервісними підходами. Крім

того, SOA вважається певною мірою застарілим підходом для побудови високонавантажених cloud-native систем.

У контексті даного проєкту SOA розглядається як надлишкове та менш ефективне рішення порівняно із сучасною мікросервісною архітектурою, яка краще відповідає вимогам масштабованості, продуктивності та гнучкості системи.

Мікросервісна архітектура (Рис. 1.7) передбачає поділ системи на набір незалежних сервісів, кожен з яких відповідає за окрему бізнес-функцію та може бути розгорнутий і масштабований автономно [7], [9].

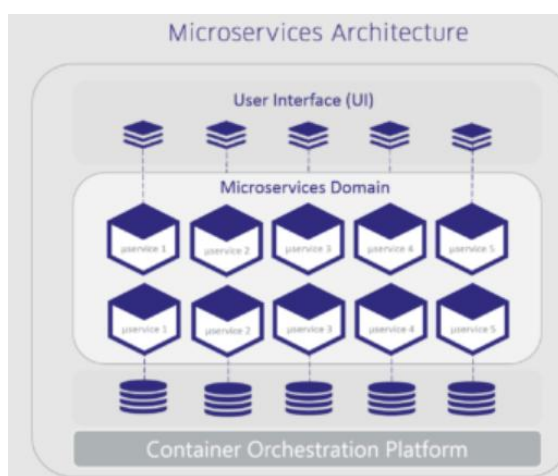


Рис 1.7. - Візуальне зображення мікросервісної архітектури

У межах даного проєкту система генерації скріншотів складається з низки основних компонентів, кожен з яких виконує чітко визначену функцію.

API Gateway (ASP.NET Core) є єдиною точкою входу для клієнтських запитів і реалізує REST API [26]. До його функціональності також належать модулі автентифікації та авторизації користувачів [30], управління API-токенами, а також базова логіка підписок і points-based тарифікації. Крім того, цей компонент відповідає за прийом запитів на генерацію скріншотів, первинну валідацію вхідних даних та постановку задач у чергу виконання.

Worker Service відповідає за безпосередню генерацію скріншотів із використанням Playwright у headless-режимі. Сервіс отримує задачі з черги повідомлень, виконує рендеринг вебсторінок відповідно до заданих параметрів і повертає результат у вигляді зображення.

Message Broker (RabbitMQ) використовується для організації асинхронної обробки задач генерації скріншотів. Він забезпечує механізм чергування запитів і дозволяє масштабувати кількість worker-інстансів, ефективно розподіляючи навантаження між ними та підвищуючи відмовостійкість системи [5], [6].

Database (PostgreSQL) застосовується для зберігання даних користувачів, метаданих скріншотів, транзакцій, логів, статусів виконання задач, а також інформації, необхідної для тарифікації та контролю використання системи.

Storage (Azure Blob Storage) використовується як об'єктне сховище для збереження згенерованих скріншотів. У реляційній базі даних при цьому зберігаються лише метадані та посилання на файли, що дозволяє зменшити навантаження на базу даних і підвищити ефективність роботи системи.

Logging Service / DB logging layer реалізує підсистему логування на основі окремих таблиць у PostgreSQL. Вона використовується для збереження помилок, діагностичної інформації та подій аудиту, що виникають у процесі роботи системи.

Загальна архітектура системи характеризується такими основними властивостями, як незалежне розгортання сервісів, асинхронна взаємодія між компонентами, використання message broker (RabbitMQ) для організації черг задач, можливість горизонтального масштабування worker-інстансів, а також чітка ізоляція відповідальностей між сервісами.

Система забезпечує високу масштабованість і відмовостійкість, підтримує незалежний розвиток окремих компонентів, ефективно обробляє великі обсяги запитів та дозволяє гнучко розподіляти навантаження за рахунок чергування задач. Водночас до недоліків такого підходу належать складність інфраструктури, необхідність використання засобів оркестрації Docker та Kubernetes, ускладненість моніторингу та логування, а також додаткові затримки, пов'язані з мережевою взаємодією між сервісами.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		24

У межах мікросервісної архітектури окремі компоненти системи реалізуються з використанням практичного модульного підходу. Замість повноцінної Clean Architecture, яка є надлишковою для даного домену, застосовується спрощена комбінація перевірених патернів, зокрема Dependency Injection (IoC) для інверсії залежностей в ASP.NET Core, Repository Pattern для абстракції доступу до даних, Result Pattern для уніфікованої обробки результатів операцій, а також підхід Feature Modules / Modular Monolith для організації коду всередині сервісів. Додатково використовується Shared Kernel для спільних моделей та утиліт між модулями.

Для взаємодії між компонентами системи застосовуються різні механізми залежно від сценарію використання. RabbitMQ використовується для асинхронної обробки задач генерації скріншотів, gRPC - для швидкої синхронної взаємодії між API Gateway та worker-сервісами (наприклад, отримання статусу або результату виконання) [14], а REST API - для зовнішньої взаємодії з клієнтськими застосунками.

Зберігання даних організовано відповідно до принципу розділення відповідальностей: PostgreSQL використовується для зберігання користувачів, метаданих скріншотів і транзакцій [2], Azure Blob Storage - для зберігання зображень [12], а RabbitMQ - як механізм чергування задач. Worker-сервіси передбачають горизонтальне масштабування, зокрема в перспективі через Kubernetes autoscaling, що дозволяє адаптувати систему до змінного навантаження.

Таким чином, мікросервісна архітектура є найбільш доцільною для даної системи, оскільки забезпечує необхідний рівень масштабованості, відмовостійкості та продуктивності. Додатково внутрішня структура сервісів базується на спрощеному модульному підході з використанням перевірених архітектурних патернів, що дозволяє уникнути надмірної складності Clean Architecture, зберігаючи при цьому високу читабельність і підтримуваність коду.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		25

1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Вибір інструментальних засобів та технологій є критично важливим етапом проєктування системи генерації скріншотів вебсторінок, оскільки він безпосередньо впливає на продуктивність, масштабованість, стабільність та подальшу підтримку програмного продукту. Зважаючи на SaaS характер системи та наявність високонавантаженої асинхронної обробки запитів, обраний стек технологій орієнтований на ефективну роботу в розподіленому середовищі.

У якості системи управління базами даних було обрано реляційну СУБД PostgreSQL [28]. Дане рішення обумовлене високим рівнем надійності, стабільності та продуктивності, що є критично важливим для системи, яка обробляє велику кількість транзакцій та задач генерації скріншотів. PostgreSQL підтримує складні запити, транзакційність та механізми забезпечення цілісності даних [29]. Окремою перевагою є підтримка розширених типів даних та індексації, що дозволяє ефективно працювати з метаданими скріншотів, користувачами та транзакціями.

Додатково варто відзначити активну спільноту та відкритий вихідний код, що забезпечує швидкий розвиток системи та високу доступність рішень для інтеграції.

Для реалізації серверної частини застосунку було обрано мову програмування C# та платформу .NET 10 [27], що забезпечує високу продуктивність, кросплатформеність та широкий набір вбудованих інструментів для розробки веб-застосунків. C# є об'єктно-орієнтованою мовою програмування, що добре інтегрується з обраною реляційною базою даних та дозволяє будувати структуровані та підтримувані рішення [3]. Використання платформи .NET також забезпечує підтримку асинхронного програмування, що є критично важливим для обробки великої кількості одночасних запитів.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		26

Для побудови серверної частини API обрано фреймворк ASP.NET Core [3], який є високопродуктивним рішенням для розробки вебзастосунків і REST API. Він забезпечує високу швидкість обробки HTTP-запитів, підтримку кросплатформного розгортання, вбудовану систему dependency injection, розширюваність та модульність, а також тісну інтеграцію з екосистемою .NET. Завдяки цим характеристикам ASP.NET Core є оптимальним вибором для реалізації API Gateway у межах мікросервісної архітектури.

Для роботи з базою даних використовується Entity Framework Core [4], який надає високорівневу абстракцію над SQL-запитами. На відміну від легковагових ORM, він дозволяє працювати з даними у вигляді об'єктів у межах підходу Code First. Це зменшує обсяг ручного SQL-коду, знижує ймовірність помилок і прискорює процес розробки.

Асинхронна взаємодія між компонентами системи реалізується за допомогою RabbitMQ, який працює на основі протоколу AMQP. Його використання дозволяє організувати черги задач генерації скріншотів, забезпечити балансування навантаження між worker-сервісами, підвищити відмовостійкість системи, а також реалізувати механізми повторних спроб (retry) у разі збоїв.

На початковому етапі інтеграція з RabbitMQ здійснюється за допомогою бібліотеки RabbitMQ.Client [5], яка надає низькорівневий доступ до функціональності брокера повідомлень. У перспективі передбачається перехід на використання MassTransit, що забезпечує більш високорівневу абстракцію та спрощує реалізацію складних сценаріїв взаємодії між сервісами, дозволяючи зосередитися на бізнес-логіці замість інфраструктурного коду.

Для безпосередньої генерації скріншотів використовується Playwright [13], який забезпечує керування headless-браузерами та високу точність рендерингу вебсторінок. Worker-сервіси, що базуються на цьому інструменті,

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		27

виконують обробку задач у фоновому режимі, що дозволяє масштабувати систему горизонтально відповідно до навантаження.

Зберігання згенерованих скріншотів реалізовано за допомогою Azure Blob Storage [12], який забезпечує високу доступність, масштабованість, економічність зберігання великих обсягів даних, а також швидкий доступ до об'єктів через HTTP. У базі даних при цьому зберігаються лише метадані та посилання на файли, що дозволяє зменшити навантаження на реляційну систему.

Для зберігання логів використовується окрема таблиця в PostgreSQL [2], що дозволяє фіксувати помилки, системні події та інформацію про виконання задач. Такий підхід забезпечує простоту реалізації та достатній рівень контролю на початкових етапах розвитку системи.

Клієнтська частина системи реалізується з використанням TypeScript [20], що забезпечує статичну типізацію, підвищує надійність коду та зменшує кількість помилок на етапі розробки. Для побудови інтерфейсу використовується React [21], який базується на компонентному підході та дозволяє створювати перевикористовувані UI-компоненти, що спрощує масштабування та підтримку застосунку.

Управління станом реалізовано за допомогою Redux [22], який забезпечує централізоване та передбачуване зберігання даних, необхідних для відображення скріншотів, статусів задач та інформації про користувача. Для реалізації асинхронної логіки використовується зв'язка RxJS та Redux-Observable [23], що дозволяє організувати реактивну обробку потоків подій, таких як HTTP-запити, polling статусів та UI-взаємодії, а також керувати побічними ефектами через епіки.

Для керування репозиторієм та організації процесу розробки використовується GitLab [25], який забезпечує відстеження задач (Issues), управління етапами розробки (Milestones), організацію робочих процесів команди та потенційну автоматизацію CI/CD-процесів.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		28

Таким чином, обраний технологічний стек забезпечує оптимальний баланс між продуктивністю, масштабованістю та простотою підтримки, що є критично важливим для SaaS-системи генерації скріншотів вебсторінок.

Висновки до першого розділу

У першому розділі було сформульовано задачу роботи та визначено основні вимоги до розроблюваної системи генерації скріншотів вебсторінок.

Було проведено аналіз існуючих рішень та сформовано порівняльну таблицю, що дозволило визначити ключові функціональні можливості та недоліки аналогічних систем, які враховано при проєктуванні власного програмного продукту.

Також виконано аналіз та вибір архітектурного підходу. В результаті обґрунтовано використання мікросервісної архітектури з модульною організацією внутрішніх сервісів, що базується на принципах спрощеної архітектурної декомпозиції та використанні перевірених патернів проєктування.

Окрім цього, було обрано технологічний стек для реалізації системи. Для серверної частини використано C#, ASP.NET Core, Entity Framework Core, RabbitMQ та PostgreSQL, що забезпечує надійну обробку запитів, асинхронну взаємодію та ефективне зберігання даних. Клієнтська частина реалізується з використанням React, Redux, RxJS та Redux-Observable, що дозволяє побудувати реактивний інтерфейс із керованою асинхронною логікою та централізованим управлінням станом.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		29

РОЗДІЛ 2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Визначення варіантів використання та архітектура сервісу

Перед початком проєктування програмної системи важливим етапом є визначення функціональних можливостей, які вони повинні надавати кінцевим користувачам. Для цього формується перелік варіантів використання (use cases), що описують взаємодію між акторами системи та її функціоналом.

У межах розроблюваної системи генерації скріншотів із підтримкою тарифікації та підписок було виділено такі основні актори (Рис 2.1):

1. Гість (Anonymous user) - користувач без авторизації;
2. Авторизований користувач - користувач із обліковим записом та доступом до основного функціоналу системи;
3. Адміністратор - користувач, що має доступ до адміністративних та аналітичних функцій системи.

Гість має обмежений доступ до системи та може лише переглядати публічну інформацію, таку як головна сторінка, опис сервісу, документація API та умови використання.

Авторизований користувач є основним актором системи та має доступ до створення та управління скріншотами, а також до функціоналу тарифікації, підписок і платіжних операцій. Він може ініціювати генерацію скріншотів за URL-адресою, налаштовувати параметри рендерингу (cookies, headers, viewport, формат зображення), а також отримувати результати виконання задач.

Адміністратор відповідає за контроль стану системи, моніторинг користувачів, перегляд логів, статистики та загального стану сервісів, що дозволяє забезпечувати стабільність і контроль роботи платформи.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		30

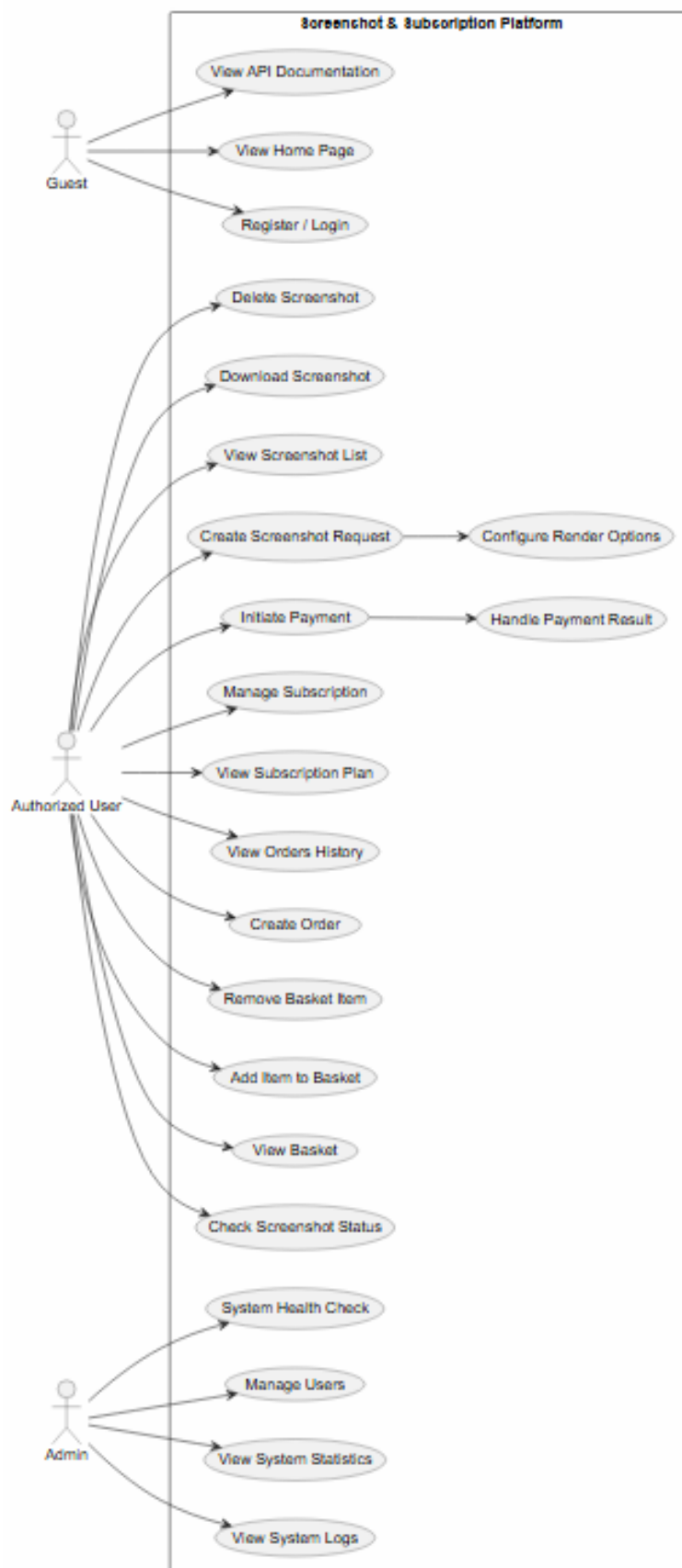


Рис 2.1. - Діаграма використання

Таблиця 2.1

Перегляд головної сторінки

Назва	Перегляд головної сторінки
Актор	Гість
Опис	Відображення загальної інформації про сервіс

Таблиця 2.2

Перегляд документації API

Назва	Перегляд головної сторінки
Актор	Гість
Опис	Ознайомлення з можливостями API

Таблиця 2.3

Реєстрація / Вхід

Назва	Перегляд головної сторінки
Актор	Гість
Опис	Створення акаунту або вхід у систему

Таблиця 2.4

Створення скріншоту

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Формування запиту на генерацію скріншоту

Таблиця 2.5

Налаштування рендерингу

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Налаштування параметрів скріншоту таких як: Cookies, headers, viewport, формат скріншоту (png/jpeg).

Таблиця 2.6

Перевірка статусу генерації скріншоту

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Користувач може перевіряти стан виконання задачі генерації скріншоту (в черзі, в обробці, завершено, помилка).

Таблиця 2.7

Перегляд тарифного плану та доступних лімітів

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Користувач може переглядати свій активний тарифний план, кількість доступних запитів/скріншотів та обмеження відповідно до підписки

Таблиця 2.8

Перегляд історії скріншотів

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Перегляд списку всіх створених користувачем скріншотів із можливістю фільтрації за статусом виконання, датою створення та URL-джерелом

Таблиця 2.9

Завантаження скріншоту

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Отримання файлу скріншоту з хмарного сховища та завантаження як картинку на комп'ютер користувача.

Таблиця 2.10

Видалення скріншоту

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Видалення файлу скріншоту з хмарного сховища та його мета даних з бази даних.

Таблиця 2.11

Генерація API token

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Генерація токену який дає доступ до певних функцій. Наприклад, токен яким можна тільки робити скріншоти, але не можна їх отримувати, що ідеально підходить для інтеграцій у тестування

Таблиця 2.12

Керування профілем

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Надання користувачу можливості переглядати та оновлювати особисті дані акаунту, включаючи базову інформацію та налаштування облікового запису

Таблиця 2.13

Перегляд логів

Назва	Перегляд головної сторінки
Актор	Адміністратор
Опис	Перегляд журналів подій системи для аналізу помилок, відстеження виконання операцій та діагностики роботи сервісів

Таблиця 2.14

Перегляд статистики

Назва	Перегляд головної сторінки
Актор	Адміністратор
Опис	Отримання агрегованих метрик роботи системи, включаючи кількість створених скріншотів, частоту помилок, навантаження на worker-сервіси та загальну продуктивність

Таблиця 2.15

Керування користувачами

Назва	Перегляд головної сторінки
Актор	Адміністратор
Опис	Адміністратор має можливість переглядати список користувачів системи та виконувати операції керування, включаючи блокування або перегляд активності

Таблиця 2.16

Перевірка стану системи (System Health Check)

Назва	Перегляд головної сторінки
Актор	Адміністратор
Опис	Надання адміністративного доступу до інформації про стан системи, включаючи доступність API Gateway, черги, бази даних та хмарного сховища картинок

Таблиця 2.17

Створення скріншоту через API

Назва	Перегляд головної сторінки
Актор	Адміністратор
Опис	Ініціація процесу генерації скріншоту за заданим URL із параметрами рендерингу (розмір, формат, режим виконання). Після створення задачі користувач отримує ідентифікатор для відстеження результату

Таблиця 2.18

Керування кошиком (Basket)

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Додавання та видалення товарів або тарифних пакетів у кошик перед оформленням замовлення, а також зміна їх кількості

Таблиця 2.19

Оформлення замовлення

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Формування замовлення на основі вмісту кошика з розрахунком загальної вартості, валюти та фіксацією переліку вибраних позицій

Проведення оплати

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Ініціація оплати замовлення через зовнішнього платіжного провайдера з фіксацією статусу транзакції (успішна, відхилена, в обробці)

Таблиця 2.21

Керування підпискою

Назва	Перегляд головної сторінки
Актор	Авторизований користувач
Опис	Активація, продовження або перегляд поточного стану підписки користувача, включаючи тип тарифного плану та дату завершення активного періоду

Після аналізу можливих сценаріїв використання системи генерації скріншотів та керування підписками (Таблиця 2.1 - 2.21) було визначено основні потреби користувачів, які необхідно врахувати при проєктуванні та реалізації програмного продукту. На основі цього сформовано вимоги до системи.

Вимоги користувачів охоплюють широкий спектр функціональності, необхідної для роботи з платформою. Зокрема, користувач повинен мати можливість переглядати головну сторінку із загальною інформацією про сервіс та тарифні плани, а також ознайомлюватися з API-документацією для інтеграції системи у зовнішні рішення. Передбачається підтримка реєстрації та авторизації користувачів для доступу до функціоналу платформи. Основний функціонал включає створення запитів на генерацію скріншотів за

заданим URL, налаштування параметрів рендерингу сторінки (viewport, cookies, headers, формат зображення), перегляд списку створених скріншотів із відображенням їхніх статусів, а також можливість завантаження та видалення згенерованих зображень із системи та хмарного сховища. Додатково система повинна забезпечувати механізми керування комерційною частиною платформи, зокрема формування та управління кошиком перед створенням замовлення, оформлення замовлень на основі вибраних сервісів або підписок, перегляд історії замовлень та їхніх статусів, а також управління активною підпискою і доступними тарифними планами. Передбачається також генерація API-токенів для інтеграції та автоматизованого доступу до сервісу.

Для адміністратора системи передбачено розширені можливості, зокрема перегляд системних логів, статистики роботи сервісу, керування користувачами та моніторинг стану системи (API, база даних, черги обробки та worker-сервіси).

Функціональні вимоги передбачають реалізацію реєстрації та авторизації користувачів із використанням електронної пошти та пароля, обробку запитів на генерацію скріншотів через API або вебінтерфейс, а також асинхронну обробку задач за допомогою черг повідомлень (RabbitMQ). Система повинна підтримувати механізми керування кошиком та формування замовлень перед оплатою, обробку платежів через зовнішнього платіжного провайдера з фіксацією спроб транзакцій, а також реалізацію підпискової моделі з різними тарифними планами. Крім того, необхідним є збереження результатів генерації скріншотів у хмарному сховищі та забезпечення адміністративного моніторингу системи, включаючи логування, статистику та контроль стану сервісів.

Вимоги до сприйняття системи передбачають, що час освоєння інтерфейсу користувачем не повинен перевищувати однієї години. Час відповіді API для стандартних операцій має становити до 5 секунд, тоді як виконання складних операцій, зокрема генерації скріншота, в середньому не

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		39

повинно перевищувати 20 секунд. Інтерфейс системи має бути інтуїтивним, адаптивним та зручним у використанні.

Вимоги до надійності визначають, що середній час безвідмовної роботи системи має становити не менше 15 днів. Допустимий рівень помилок не повинен перевищувати 1 на 1000 запитів. Додатково система повинна забезпечувати відмовостійку обробку задач за рахунок використання retry-механізмів у worker-сервісах.

Вимоги до продуктивності передбачають підтримку щонайменше 100 одночасних користувачів. Масштабування системи має забезпечуватися за рахунок асинхронної обробки задач через RabbitMQ із можливістю горизонтального масштабування worker-сервісів. Також використання підходу CQRS дозволяє розділити навантаження між операціями читання та запису, підвищуючи загальну ефективність системи.

З огляду на обраний технологічний стек, ключовим завданням проєктування системи є побудова об'єктно-орієнтованої доменної моделі з урахуванням мікросервісної архітектури, асинхронної взаємодії та розділення відповідальностей між компонентами системи.

З урахуванням обраної мікросервісної архітектури, система генерації скріншотів побудована на принципі розділення відповідальностей між компонентом, що приймає та координує запити користувачів (API Gateway), та компонентом, що виконує ресурсомістку обробку (Worker-сервіс). Такий підхід дозволяє забезпечити масштабованість, ізоляцію навантаження та стабільність виконання довготривалих операцій.

Доменна модель API Gateway побудована відповідно до принципів розділення відповідальностей. Основна роль API Gateway полягає у координації бізнес-процесів, пов'язаних із користувачами, платежами, замовленнями та генерацією скріншотів, при цьому виконання ресурсомістких операцій делегується Worker-сервісу. У межах системи виділено ключові доменні сутності, які відображають основні бізнес-моделі та їх взаємозв'язки (Таблиця. 2.22):

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		40

Таблиця 2.22

Сутність	Опис
UserEntity	Центральна сутність, що представляє користувача системи. Вона слугує агрегуючим коренем для пов'язаних бізнес-даних і має зв'язки з сутностями скріншотів, кошика, замовлень, підписок, платіжних спроб, API токенів, refresh токенів.
ScreenshotEntity	Сутність скріншота. Кожен скріншот належить конкретному користувачу та може бути асоційований з множиною категорій, що реалізує зв'язок типу M:N через проміжну структуру. Додатково сутність містить метадані для зберігання стану виконання, типу скріншота та інформації для пошуку.
BasketEntity	Сутність поточного кошика користувача та має зв'язок 1:1 із користувачем. Він містить позиції кошика, які є сутністю BasketLineEntity
BasketLineEntity	Сутність що представляє позицію кошика
OrderEntity	Сутність що формується на основі кошика та представляє оформлене замовлення користувача. Воно містить набір позицій OrderLineEntity.
OrderLineEntity	Сутність що фіксує деталі кожного елемента замовлення, включаючи ціну на момент створення.
PaymentAttemptEntity	Сутність що відображає спробу проведення платежу, пов'язану із конкретним замовленням та користувачем.
SubscriptionEntity	моделює підписку користувача та визначає активний тарифний план, період дії та пов'язані ліміти використання системи.
CategoryEntity	Сутність що використовується для класифікації скріншотів, забезпечуючи гнучку систему категоризації результатів.
ApiTokenEntity	Сутність що представляє API токени доступу користувача, що використовуються для автентифікації та авторизації.
RefreshTokenEntity	Сутність що використовується для механізму оновлення сесій користувача та забезпечує безпечне продовження автентифікації.
LogEntity	Сутність що використовується для системного журналювання подій, помилок та діагностичної інформації.
AdminEntity	Сутність що представляє адміністратора системи

Таким чином, доменна модель API Gateway (Рис. 2.2) забезпечує чітке відображення бізнес-логіки системи та підтримує масштабовану структуру взаємодії між користувацькими, платіжними та обчислювальними процесами.

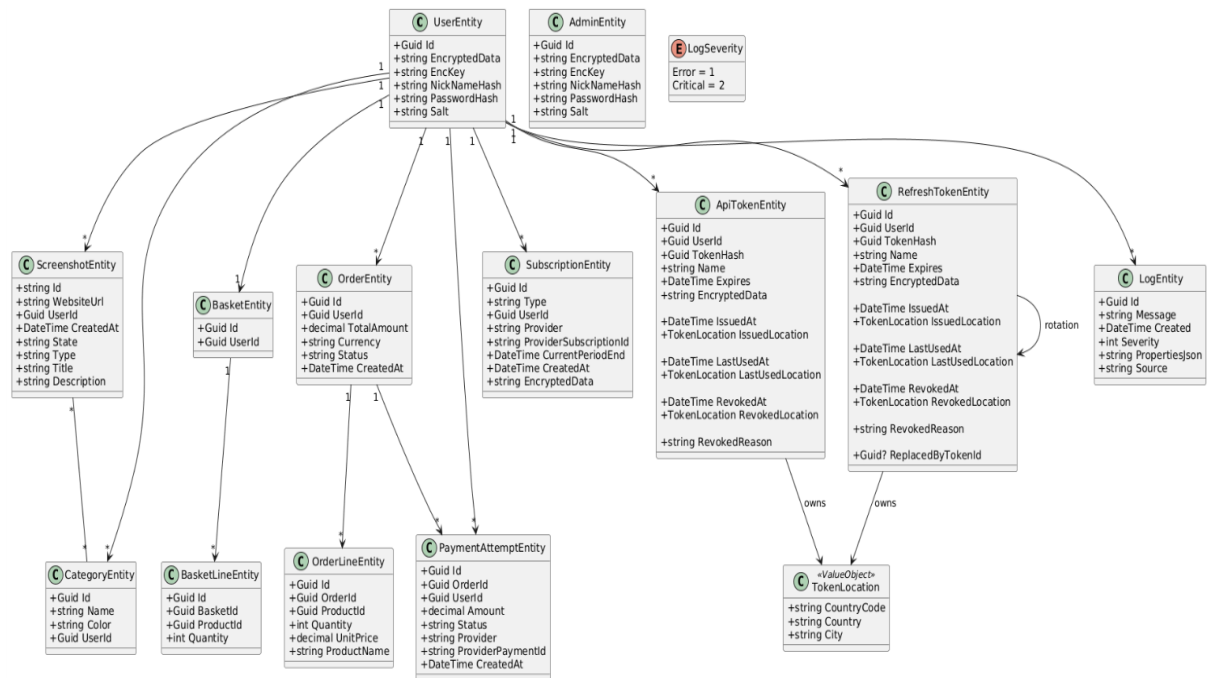


Рис 2.2. - Діаграма залежностей сутностей

Worker Service відповідає за виконання ресурсомістких операцій генерації скріншотів, які характеризуються високим навантаженням на CPU та оперативну пам'ять. Його архітектура побудована за принципом багаторівневого pipeline з чітким розподілом відповідальностей між компонентами, що забезпечує масштабованість, стабільність та контроль використання ресурсів. Уся система Worker-сервісу складається з кількох логічно ізольованих підсистем, кожна з яких виконує окрему роль у процесі обробки задач (рис. 2.3).

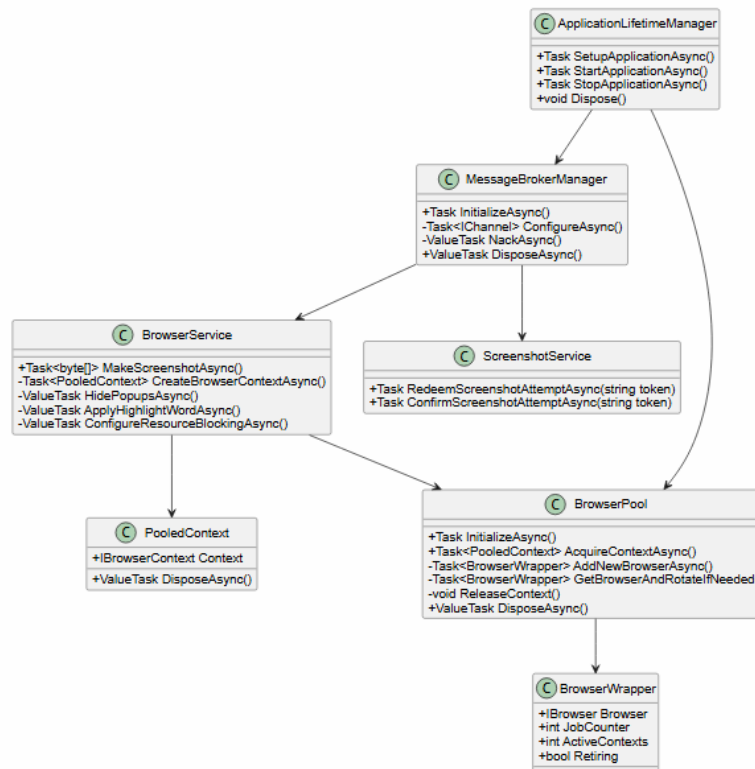


Рис 2.3. - Діаграма залежності сервісів у worker сервісі

ApplicationLifetimeManager відповідає за життєвий цикл Worker-сервісу. Він забезпечує ініціалізацію середовища виконання, включаючи перевірку наявності Chromium, запуск браузерного пулу та підключення до брокера повідомлень. Окрім цього, компонент координує коректне завершення роботи сервісу та звільнення ресурсів.

MessageBrokerManager реалізує інтеграцію з RabbitMQ та виконує роль споживача черги задач. Його основними обов'язками є отримання повідомлень, їх десеріалізація, керування підтвердженням обробки (ACK/NACK), а також передача валідних задач до бізнес-рівня обробки. Таким чином, він виступає вхідною точкою для всіх задач Worker-сервісу.

BrowserService є основним виконавчим компонентом, що реалізує бізнес-логіку генерації скріншотів. Він відповідає за створення браузерних сторінок, навігацію до цільових URL, застосування параметрів рендерингу (headers, cookies, viewport), а також додаткову обробку сторінки, включаючи

приховування елементів інтерфейсу, підсвічування тексту та блокування ресурсів. Результатом його роботи є сформоване зображення вебсторінки.

BrowserPool виконує функцію управління браузерними ресурсами та оптимізації продуктивності. Він реалізує механізм повторного використання браузерних інстансів через пулізацію, контролює кількість одночасних контекстів, здійснює ротацію браузерів після виконання певної кількості задач, а також забезпечує ізоляцію виконання через механізми синхронізації. Це дозволяє стабілізувати використання пам'яті та запобігати деградації продуктивності під час тривалої роботи системи.

ScreenshotService реалізує gRPC-клієнт для комунікації з API Gateway. Він використовується для передачі результатів виконання задачі, зокрема підтвердження успішної генерації скріншоту або фіксації помилки виконання. Такий підхід забезпечує асинхронну двосторонню взаємодію між сервісами та дозволяє відокремити процес генерації від бізнес-логіки API Gateway.

У сукупності дана архітектура забезпечує ефективну обробку великої кількості паралельних задач, ізоляцію ресурсомістких операцій та стабільну інтеграцію між сервісами через черги повідомлень та gRPC-виклики.

Архітектура Worker-сервісу побудована за принципом послідовного pipeline (рис. 2.4), у межах якого кожен компонент виконує чітко визначену роль. Зокрема, MessageBrokerManager відповідає за отримання повідомлень із черги, BrowserService реалізує бізнес-логіку генерації скріншотів, а BrowserPool здійснює управління браузерними ресурсами та їх оптимізацію.

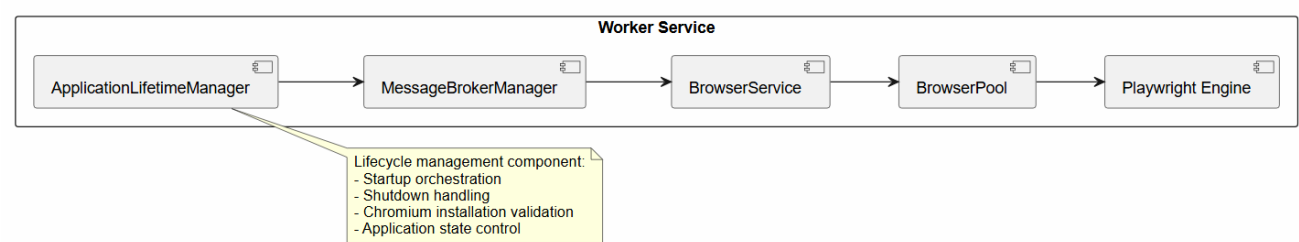


Рис 2.4. - Діаграма архітектури worker сервісу

У системі реалізовано механізм повторного використання браузерних контекстів, що дозволяє зменшити накладні витрати на ініціалізацію Chromium. Такий підхід забезпечує також контроль кількості паралельно виконуваних задач і гарантує ізоляцію виконання через використання окремих контекстів для кожного запиту.

Для забезпечення стабільності довготривалої роботи системи браузерні інстанси періодично перезапускаються після виконання визначеної кількості задач. Це дозволяє запобігати витокам пам'яті, підтримувати стабільність продуктивності та очищати внутрішній стан Chromium.

Механізм SemaphoreSlim використовується для обмеження кількості одночасно активних браузерних контекстів. Це дозволяє уникати перевантаження CPU та оперативної пам'яті, а також забезпечує передбачувану продуктивність системи навіть при високому навантаженні.

RabbitMQ виступає основним механізмом чергування задач у системі. Його використання забезпечує декуплінг API Gateway і Worker-сервісів, можливість горизонтального масштабування worker-інстансів, а також гарантує доставку повідомлень і стабільну обробку запитів у асинхронному режимі.

gRPC використовується виключно як механізм зворотного зв'язку між сервісами. Його основними задачами є підтвердження успішного виконання задач, передача інформації про помилки, а також мінімізація затримок при синхронізації станів між API Gateway та Worker-сервісами.

У результаті запропонована архітектура забезпечує слабку зв'язаність компонентів, ефективне використання ресурсів та можливість масштабування системи відповідно до зростання навантаження.

2.2 Розробка бази даних

Система, що розробляється, побудована на основі мікросервісної архітектури, що передбачає логічне та фізичне розділення компонентів. Одним із ключових принципів такого підходу є використання окремих баз даних для кожного мікросервісу. Це дозволяє забезпечити ізоляцію даних,

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		45

підвищити незалежність розгортання сервісів та спростити масштабування системи.

У межах розроблюваної системи генерації скріншотів основним компонентом, що працює з базою даних, є API Gateway, який відповідає за збереження бізнес-даних, пов'язаних із користувачами, скріншотами, замовленнями, платежами та підписками. Worker Service, у свою чергу, не має власної бази даних і виконує виключно обчислювальні задачі, що дозволяє уникнути дублювання стану та спростити архітектуру. Таким чином, база даних API Gateway є центральним сховищем доменних даних системи та забезпечує узгодженість бізнес-процесів. База даних API Gateway реалізована за допомогою Entity Framework Core та включає набір таблиць, що відповідають доменним сутностям системи. До складу бази даних входять такі таблиці (Таблиця 2.23):

Таблиця 2.23

Таблиця	Опис
Users	користувачі системи
Admins	адміністратори системи
Screenshots	результати генерації скріншотів
Categories	категорії для класифікації скріншотів;
Baskets	кошики користувачів;
BasketLines	позиції кошика;
Orders	замовлення;
OrderLines	позиції замовлення;
PaymentAttempts	спроби оплати;
Subscriptions	підписки користувачів
ApiTokens	API токени доступу;
RefreshTokens	токени оновлення сесії;
Logs	системні журнали подій;
__EFMigrationsHistory	службова таблиця історії міграцій.

Таблиця 2.24

Опис таблиці “Users”

Поле	Тип	Призначення
Id	Uuid	Унікальний ідентифікатор користувача
EncryptedData	Text	Зашифровані дані користувача
EncKey	varchar(128)	Зашифрований мастер лючем ключ шифрування
NickNameHash	varchar(128)	Хеш нікнейму
PasswordHash	varchar(128)	Хеш пароля
Salt	varchar(128)	Сіль для хешування

Таблиця 2.25

Опис таблиці “Screenshots”

Поле	Тип	Призначення
Id	varchar(32)	Ідентифікатор скріншоту
WebsiteUrl	varchar(2048)	URL веб-сторінки
UserId	Uuid	Ідентифікатор користувача
CreatedAt	Timestamp	Дата створення
State	varchar(20)	Статус виконання
Type	varchar(20)	Тип зображення
Title	varchar(150)	Заголовок скріншоту
Description	varchar(500)	Опис

Додатково реалізовано індекси для прискорення пошуку, підтримку повнотекстового пошуку на основі PostgreSQL (GIN + tsvector), а також зв'язок типу багато-до-багатьох із таблицею Categories.

Таблиця 2.26

Опис таблиці “Categories”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор категорії
Name	varchar(100)	Назва категорії
Color	varchar(20)	Колір
UserId	Uuid	Власник категорії

Таблиця 2.27

Опис таблиці “Baskets”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор баскету
UserId	Uuid	Ідентифікатор користувача

Таблиця 2.28

Опис таблиці “BasketLines”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор баскет лінії
BasketId	Uuid	Ідентифікатор користувача
ProductId	Uuid	Ідентифікатор продукту
Quantity	Integer	Кількість продукту

Таблиця 2.29

Опис таблиці “Orders”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор замовлення
UserId	Uuid	Ідентифікатор користувача
TotalAmount	Decimal	Загальна сума замовлення
Currency	varchar(10)	Валюта
Status	varchar(20)	Статус оплати
CreatedAt	Timestamp	Дата створення

Таблиця 2.30

Опис таблиці “OrderLines”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор лінії замовлення
OrderId	Uuid	Ідентифікатор замовлення
ProductId	Uuid	Ідентифікатор продукту
Quantity	Integer	Кількість
UnitPrice	varchar(20)	Ціна
ProductName	varchar(100)	Назва продукту (є сенс зберігати, бо з часом назва продукту може змінитись або він взагалі буде видалений)

Таблиця 2.31

Опис таблиці “PaymentAttempts”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор транзакції

продовж. табл. 2.31

OrderId	Uuid	Ідентифікатор замовлення
UserId	Uuid	Ідентифікатор користувача
Amount	Decimal	Сума
Status	varchar(20)	Статус
Provider	varchar(50)	Платіжний провайдер
ProviderPaymentId	varchar(200)	ID транзакції
CreatedAt	Timestamp	Дата створення

Таблиця 2.32

Опис таблиці “Subscriptions”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор підписки
Type	varchar(20)	Тип підписки
UserId	Uuid	Ідентифікатор користувача
Provider	varchar(50)	Платіжний провайдер
ProviderSubscriptionId	varchar(200)	ID підписки на стороні платіжного провайдера
CurrentPeriodEnd	Timestamp	Кінець підписки
CreatedAt	Timestamp	Дата створення
EncryptedData	Text	Зашифровані дані підписки (включає у себе платіжний токен)

Опис таблиці “ApiTokens”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор токена
UserId	varchar(20)	Власник токена
TokenHash	Uuid	Хеш токена
Name	varchar	Назва токена
Expires	timestamp	Термін дії
EncryptedData	text	Метадані токена
TokenMetadata_Issued	timestamp	Дата створення токена
TokenMetadata_IssuedLocation_CountryCode	varchar	Код країни з якої було створено токен
TokenMetadata_IssuedLocation_Country	varchar	Назва країни з якої було створено токен
TokenMetadata_IssuedLocation_City	varchar	Код міста з якого було створено токен
TokenMetadata_LastUsed	timestamp	Дата останнього використання токена
TokenMetadata_LastUsedLocation_CountryCode	varchar	Код країни з якої було використано токен
TokenMetadata_LastUsedLocation_Country	varchar	Назва країни з якої було використано токен
TokenMetadata_LastUsedLocation_City	varchar	Код міста з якого було використано токен
TokenMetadata_Revoked	timestamp	Дата відкликання токена

продовж. табл. 2.33

TokenMetadata_Revoked Location_CountryCode	varchar	Код країна з якої було відкликано токен
TokenMetadata_Revoked Location_Country	varchar	Назва країна з якої було відкликано токен
TokenMetadata_Revoked Location_City	varchar	Код міста з якого було відкликано токен
RevokedReason	varchar	Причина відкликання токена

Таблиця 2.34

Опис таблиці “RefreshTokens”

Поле	Тип	Призначення
Id	Uuid	Ідентифікатор токена
UserId	varchar(20)	Власник токена
TokenHash	Uuid	Хеш токена
Name	varchar	Назва токена
Expires	timestamp	Термін дії
EncryptedData	text	Метадані токена
TokenMetadata_Issued	timestamp	Дата створення токена
TokenMetadata_IssuedLocation_CountryCode	varchar	Код країна з якої було створено токен
TokenMetadata_IssuedLocation_Country	varchar	Назва країна з якої було створено токен
TokenMetadata_IssuedLocation_City	varchar	Код міста з якого було створено токен
TokenMetadata_LastUsed	timestamp	Дата останнього використання токена

продовж. табл. 2.34

TokenMetadata_LastUsedLocation_CountryCode	varchar	Код країна з якої було використано токен
TokenMetadata_LastUsedLocation_Country	varchar	Назва країна з якої було використано токен
TokenMetadata_LastUsedLocation_City	varchar	Код міста з якого було використано токен
TokenMetadata_Revoked	timestamp	Дата відкликання токена
TokenMetadata_RevokedLocation_CountryCode	varchar	Код країна з якої було відкликано токен
TokenMetadata_RevokedLocation_Country	varchar	Назва країна з якої було відкликано токен
TokenMetadata_RevokedLocation_City	varchar	Код міста з якого було відкликано токен
RevokedReason	varchar	Причина відкликання токена
ReplacedByTokenId	uuid	ID токена, яким замінено (rotation)

Таблиця 2.35

Опис таблиці “Admins”

Поле	Тип	Призначення
Id	Uuid	Унікальний ідентифікатор адміністратора
EncryptedData	Text	Зашифровані дані користувача
EncKey	varchar(128)	Зашифрований мастер лючем ключ шифрування
NickNameHash	varchar(128)	Хеш нікнейму

продовж. табл. 2.35

PasswordHash	varchar(128)	Хеш пароля
Salt	varchar(128)	Сіль для хешування

Таблиця 2.36

Опис таблиці “Logs”

Поле	Тип	Призначення
Id	UUID	Ідентифікатор логу
Message	Text	Повідомлення логу
Created	timestamp	Дата та час створення логу
Severity	INT (enum: 1=Error, 2=Critical)	Серйозність проблеми яку було залоговано
PropertiesJson	JSON	Додаткові параметри логу
Source	varchar(128)	Сервіс, що створив лог

Таблиця 2.37

Опис таблиці “_EFMigrationsHistory”

Поле	Тип	Призначення
MigrationId	Varchar	Ідентифікатор міграції
ProductVersion	Varchar	Версія EF Core

У процесі проектування бази даних (Таблиця 2.24 - 2.37) було прийнято низку ключових рішень, спрямованих на продуктивність, цілісність даних і масштабованість. Використання Entity Framework Core забезпечує узгодженість доменної моделі з базою даних, а індексація критичних полів підвищує швидкість доступу до даних. Додатково реалізовано повнотекстовий пошук для ефективної роботи зі скріншотами та застосовано зовнішні ключі для забезпечення цілісності зв'язків.

Для відокремлення бізнес-операцій використовуються окремі таблиці транзакцій (Orders, PaymentAttempts), а система логування та аудиту реалізована через таблицю Logs, що дозволяє централізовано відстежувати події, помилки та стан системи.

Таким чином, розроблена база даних (Рис 2.5) забезпечує структуроване зберігання даних, підтримує ключові бізнес-процеси системи та дозволяє ефективно масштабувати застосунок. Використання окремої бази даних для API Gateway відповідає принципам мікросервісної архітектури та забезпечує високу гнучкість і надійність системи в цілому.

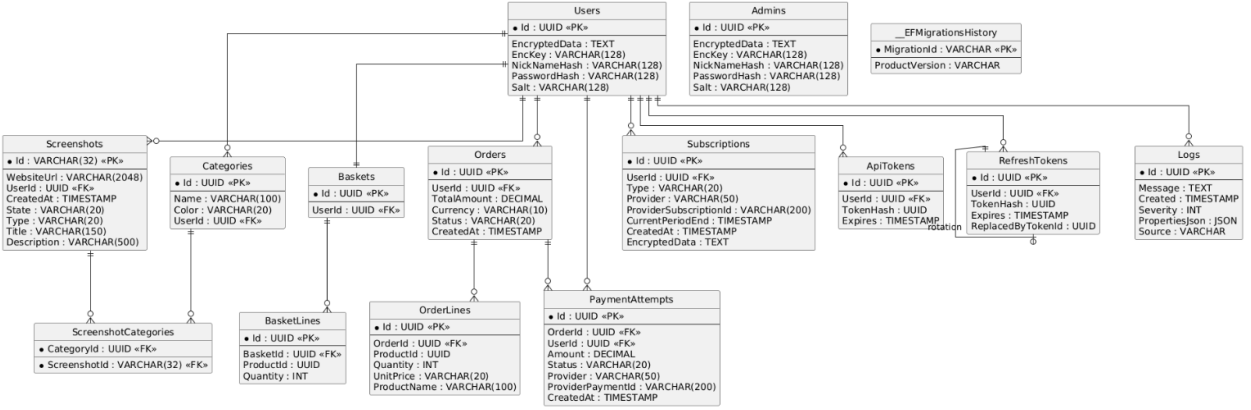


Рис 2.5 Діаграма бази даних

2.3 Проєктування та реалізація алгоритмів системи

При створенні програмної системи генерації скріншотів важливим етапом є формалізація алгоритмів обробки запитів користувача та взаємодії між компонентами системи. Це дозволяє забезпечити передбачуваність виконання операцій, контроль помилок та ефективне використання ресурсів.

Для цього було побудовано діаграми основних варіантів використання, які відображають послідовність виконання ключових бізнес-процесів. Основним сценарієм роботи системи є створення користувачем запиту на генерацію скріншоту веб-сторінки. Даний процес включає валідацію вхідних даних, створення доменної сутності та передачу задачі на асинхронну обробку.

Алгоритм починається з отримання запиту від користувача (Рис 2.6). Після цього система виконує валідацію параметрів, включаючи URL та

конфігурацію рендерингу. У разі успішної валідації виконується перевірка доступних лімітів користувача відповідно до активної підписки. Якщо ліміт не перевищено, створюється сутність скріншоту та генерується унікальний токен підтвердження. Далі задача передається у чергу повідомлень RabbitMQ для асинхронної обробки Worker-сервісом.

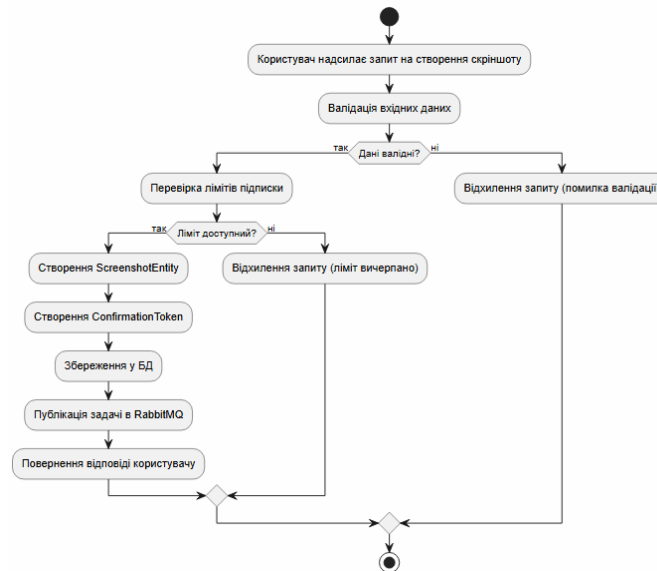


Рис 2.6. - Діаграма обробки запиту на створення скріншоту API Gateway

Другим ключовим процесом є виконання задачі генерації скріншоту Worker-сервісом (Рис 2.7).

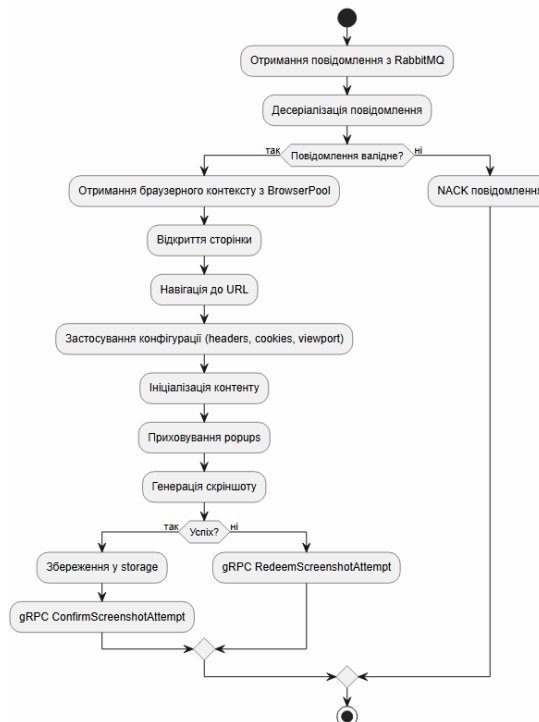


Рис 2.7. - Діаграма обробки запиту на створення скріншота Worker-сервісом

Worker-сервіс отримує повідомлення з черги та виконує його десеріалізацію. У випадку помилки повідомлення відхиляється. Далі відбувається отримання браузерного контексту з пулу (Рис 2.8), що дозволяє оптимізувати використання ресурсів. Після цього створюється сторінка, виконується навігація та застосовуються всі необхідні параметри рендерингу.

Після обробки сторінки формується скріншот. У разі успішного виконання результат зберігається, а API Gateway отримує підтвердження через gRPC. У випадку помилки виконується повідомлення про відхилення операції.Окремо було розглянуто алгоритм взаємодії між API Gateway, RabbitMQ та Worker-сервісом.

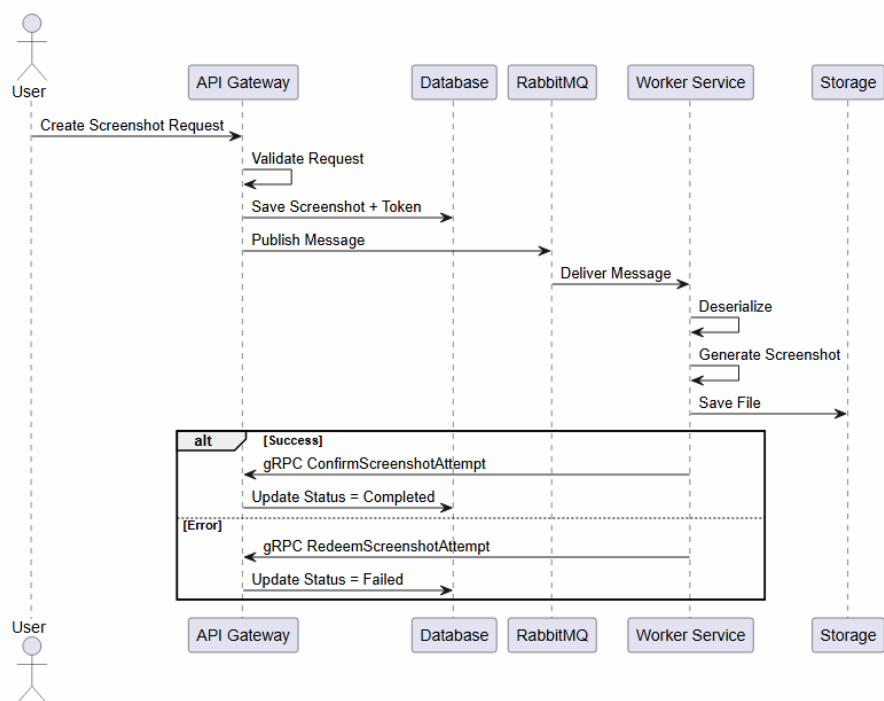


Рис 2.8. - Діаграма взаємодії API Gateway, RabbitMQ та Worker-сервісу у процесі створення скріншоту

Взаємодія починається із запиту користувача до API Gateway. Після валідації даних та створення відповідних записів у базі даних, система публікує повідомлення у RabbitMQ. Worker-сервіс асинхронно отримує задачу та виконує генерацію скріншоту. Результат зберігається у зовнішньому сховищі. Після завершення обробки Worker викликає API

Gateway через gRPC, передаючи статус виконання. Це дозволяє оновити стан задачі без блокування основного потоку обробки запитів.

2.4 Реалізація функціоналу системи генерації скріншотів вебсторінок

Головною функцією системи є генерація скріншота, яка реалізується як асинхронний багатоступеневий процес із чітким розділенням відповідальностей між API рівнем і Worker-сервісом. Такий підхід дозволяє ізолювати ресурсоємні операції рендерингу від HTTP-запитів і забезпечити горизонтальне масштабування системи.

Точкою входу є REST API контролер ScreenshotController, який приймає запит на створення скріншота через метод MakeScreenshot([FromBody] InputScreenshotModel screenshotOptions). На цьому етапі виконується автентифікація користувача через policy-based authorization (Policies.User.MakeScreenshots), після чого система отримує поточний контекст користувача через IUserContextAccessor.GetCurrentUser().

Цей контекст містить ідентифікатор користувача, інформацію про тарифний план та інші параметри сесії, необхідні для подальшої бізнес-логіки. Далі модель InputScreenshotModel трансформується у внутрішню доменну модель через screenshotOptions.ToScreenshotOptions(userContext).

Під час цього етапу виконується повна валідація параметрів, включаючи перевірку допустимих значень, коректність конфігурації рендерингу та відповідність обмеженням тарифного плану. У випадку некоректних даних система одразу повертає BadRequest, не запускаючи процес генерації, що дозволяє уникнути зайвого навантаження на інфраструктуру. Після успішної валідації контролер викликає бізнес-рівень через IScreenshotService.MakeScreenshotAsync(...), передаючи підготовлену доменну модель. Контролер при цьому не виконує жодної логіки рендерингу або взаємодії з браузером, що є ключовим елементом асинхронної архітектури системи.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		58

Основна бізнес-логіка зосереджена у ScreenshotService, зокрема в методі `MakeScreenshotAsync(ScreenshotOptionsModel screenshotOptionsModel)`. На цьому етапі сервіс виконує оркестрацію всього процесу створення скріншота. Спочатку отримується контекст користувача через `_userContextAccessor.GetCurrentUser()`, після чого генерується унікальний ідентифікатор скріншота через `MakeScreenshotId()`, який базується на Guid Version 7 для забезпечення часової впорядкованості та високої продуктивності індексації.

Далі система виконує розрахунок вартості операції у вигляді points через `IScreenshotCalculator.CalculatePoints(...)`. Цей компонент аналізує параметри рендерингу, включаючи viewport, full-page режим, роботу з DOM, використання cookies і custom headers, очікування мережевих запитів, а також додаткові опції на кшталт блокування ресурсів або очікування певних селекторів. Наприклад, full-page рендер збільшує вартість через додаткове навантаження на браузерний рушій, тоді як блокування ресурсів може зменшувати загальну складність запиту за рахунок зменшення кількості завантажуваних ресурсів.

Після розрахунку вартості виконується перевірка доступності ліміту користувача через `_subscriptionManager.ScreenshotWasMadeAsync(new(pointsCost))`. Якщо у користувача недостатньо points, процес зупиняється до моменту створення задачі, що дозволяє запобігти перевантаженню системи та неконтрольованому використанню ресурсів.

У випадку успішної перевірки генерується confirmationToken через `_authorizationManager.GenerateConfirmationToken(newConfirmationData(...))`. Цей токен містить ідентифікатор користувача, ідентифікатор скріншота, вартість операції та додаткові транзакційні дані. Він використовується Worker-сервісом для підтвердження успішного виконання задачі або виконання компенсаційних операцій у випадку помилки, зокрема повернення списаних points.

Після цього створюється запис у базі даних через `_screenshotManager.MakeAsync(...)`, де зберігаються URL, параметри рендерингу, початковий статус задачі та вартість операції. Це дозволяє забезпечити історичність, відстеження стану задачі та подальший доступ до результатів.

Наступним етапом є публікація задачі у RabbitMQ через `_messageBrokerManager.SendMessageAsync(model, routingKey)`. Повідомлення типу `MakeScreenshotModel` містить усі необхідні параметри для виконання задачі, включаючи `confirmationToken` і метадані користувача. Маршрутизація здійснюється через `routingKey`, який визначається на основі тарифного плану користувача через `_queueConfig.QueuePerSubscription[screenshotPlan.Type]`. Це дозволяє реалізувати розділення черг, пріоритетну обробку та незалежне масштабування різних класів навантаження.

Message broker реалізовано через MassTransit поверх RabbitMQ, що забезпечує підтримку retry політик, маршрутизації повідомлень та стабільної доставки. Для захисту від зависання використовується `CancellationTokenSource` із таймаутом, а передача `routingKey` додатково дублюється у headers через `context.Headers.Set(...)`. У випадку будь-якої помилки система виконує компенсаційні дії, включаючи rollback списаних points і запис помилки в лог, після чого повертає клієнту відповідну відповідь.

Worker Service є окремим мікросервісом, відповідальним за виконання ресурсоємної частини процесу - генерації скріншотів через Playwright. Його запуск координується через `ApplicationLifetimeManager`, який забезпечує перевірку та встановлення Chromium через `Microsoft.Playwright.Program.Main(["install", "chromium", "--with-deps"])`, ініціалізацію `BrowserPool` та запуск `MessageBrokerManager`. Архітектура Worker побудована навколо повторного використання браузерів через `BrowserPool`, що дозволяє зменшити overhead на створення Chromium інстансів. Для контролю одночасного виконання задач використовується

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		60

SemaphoreSlim, який обмежує кількість активних browser context і забезпечує стабільність використання CPU та RAM. Додатково реалізовано механізм ротації браузерів після досягнення певної кількості задач, що запобігає memory leaks і деградації продуктивності при тривалій роботі. Основна логіка генерації реалізована у BrowserService.

Спочатку створюється ізольований browser context через BrowserNewContextOptions, де задаються locale, timezone, JavaScript режим, cookies та viewport. Це дозволяє емулювати різні середовища користувача та гарантує повну ізоляцію між задачами. Далі виконується навігація через page.GotoAsync(screenshotOptionsModel.Url, new() { WaitUntil = WaitUntilState.DOMContentLoaded }), що забезпечує стабільний стан DOM без очікування повного завантаження ресурсів.

Після цього запускається pipeline ініціалізації контенту через ContentInitializationManager, який координує набір кроків обробки сторінки. У процесі підготовки сторінки виконується прокрутка через page.EvaluateAsync(...) для підтримки infinite scroll та SPA, очікування стабілізації мережі через page.WaitForLoadStateAsync(LoadState.NetworkIdle, ...), а також очікування появи необхідних елементів через page.WaitForSelectorAsync(...). Це дозволяє гарантувати, що скріншот буде зроблений у коректному стані UI. Додатково застосовується selective resource blocking через page.RouteAsync(...), що дозволяє блокувати CSS, JavaScript, зображення та інші ресурси для зменшення часу рендерингу і навантаження на систему. Також виконується DOM-модифікація, включаючи приховування роруп-елементів через JavaScript та можливість підсвічування контенту через HTML тег <mark>.

Фінальним етапом є генерація скріншота через page.ScreenshotAsync(...), який підтримує PNG, JPEG, full-page та element-based режими захоплення.

У результаті система реалізує повністю асинхронну, масштабовану архітектуру, де API рівень виконує лише оркестрацію та валідацію, тоді як

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		61

усі ресурсоємні браузерні операції винесені у Worker-сервіс, що забезпечує високу продуктивність, ізоляцію навантаження та стабільність роботи під високими навантаженнями.

Висновки до другого розділу

У межах другого розділу було здійснено проєктування системи генерації скріншотів вебсторінок відповідно до визначених функціональних вимог та сценаріїв використання.

На етапі аналізу варіантів використання були сформовані основні сценарії взаємодії користувача із системою та визначено загальну архітектуру рішення. Обрано мікросервісний підхід із розподілом відповідальності між API Gateway та Worker Service, що забезпечує гнучкість і масштабованість системи.

Під час розробки бази даних було спроектовано доменну модель, виділено ключові сутності та встановлено зв'язки між ними. Для реалізації доступу до даних застосовано ORM-підхід, що спростило роботу з даними та підвищило підтримуваність коду.

У межах проєктування та реалізації алгоритмів системи було розроблено механізми обробки запитів і виконання основних бізнес-процесів. Окрему увагу приділено асинхронній обробці задач із використанням RabbitMQ, а також організації міжсервісної взаємодії через gRPC, що дозволило ефективно обробляти ресурсоємні операції.

Під час реалізації функціоналу генерації скріншотів вебсторінок було впроваджено ключові компоненти серверної та інфраструктурної частини системи, які забезпечують безпосереднє виконання основної функціональності.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		62

РОЗДІЛ 3 ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ ІЗ СИСТЕМОЮ

3.1 Порядок встановлення та налаштування параметрів системи

Система реалізується на основі мікросервісного підходу, де кожен функціональний компонент - API Gateway, Worker Service, брокер повідомлень, база даних і blob storage - розгортається як окремий незалежний сервіс. Така декомпозиція дозволяє розділити відповідальність між частинами системи та масштабувати їх окремо залежно від навантаження.

У межах архітектури сервіси можуть працювати на різних контейнерах або фізичних вузлах. API Gateway виступає єдиною точкою входу та маршрутизує запити до внутрішніх компонентів. Worker Service виконує ресурсомісткі задачі, зокрема генерацію скріншотів за допомогою Playwright, що дозволяє винести важкі обчислення за межі основного потоку запитів. Для асинхронної взаємодії між сервісами використовується RabbitMQ, а PostgreSQL слугує основним сховищем структурованих даних.

Завдяки такій організації система є стійкішою до збоїв. Крім того, кожен компонент може оновлюватися окремо, що зменшує ризик простоїв і спрощує процес розгортання нових версій. Для кращого розуміння процесу розгортання було змодельовано діаграму використання (рис. 3.1)

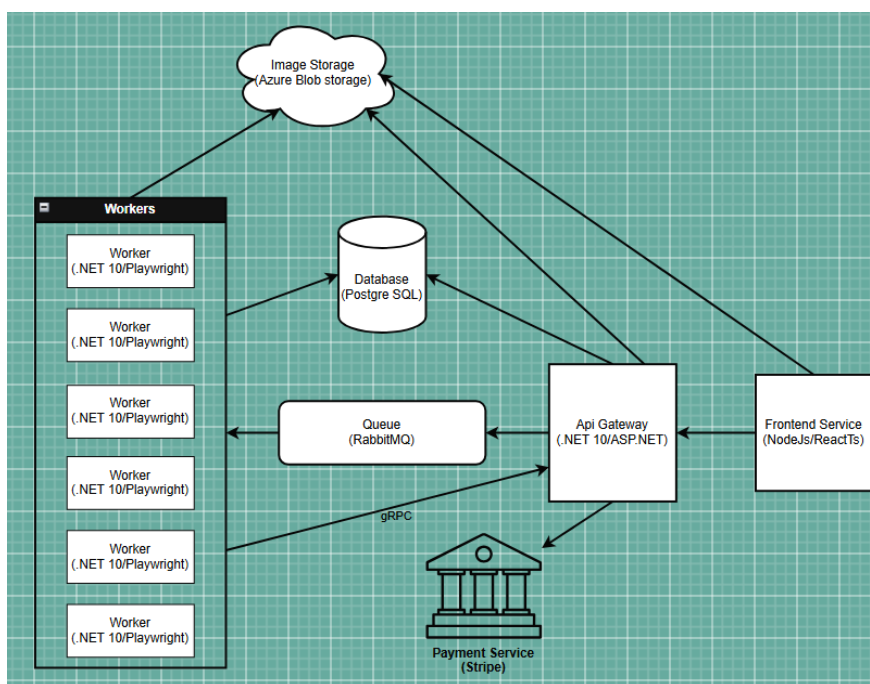


Рис. 3.1. - Діаграма розгортання

		Погойдаш І.С.		
		Єфремов Ю.М.		
Змн.	Арк.	№ докум.	Підпис	Дата

На діаграмі розгортання представлено кілька серверів, на яких розміщено основні компоненти системи: API Gateway, Worker Service для генерації скріншотів, сервери баз даних PostgreSQL, а також сервер брокера повідомлень RabbitMQ.

API Gateway виконує роль центрального координатора системи та обробляє HTTP-запити користувачів, після чого передає задачі на асинхронну обробку через RabbitMQ. Worker Service працює як окремий обчислювальний вузол і виконує ресурсоємну операцію генерації скріншотів за допомогою Playwright і Chromium. Сервери баз даних забезпечують зберігання доменних даних (користувачі, замовлення, скріншоти), а RabbitMQ забезпечує обмін повідомленнями між сервісами.

Ізоляція компонентів дозволяє уникати каскадних збоїв: відмова Worker Service не впливає на роботу API Gateway, а проблеми з брокером повідомлень не порушують цілісність збережених даних. Незалежне розгортання кожного мікросервісу дозволяє оновлювати або масштабувати окремі частини системи без зупинки всієї платформи.

Розміщення Worker Service окремо від API Gateway забезпечує ефективний розподіл ресурсів, оскільки браузерна автоматизація є обчислювально та пам'яттєво інтенсивною. Такий підхід дозволяє масштабувати лише Worker-вузли при зростанні навантаження, не змінюючи інші компоненти системи.

Розгорнути серверну частину системи необхідно на сервері з такими мінімальними характеристиками (Таблиця 3.1):

Таблиця 3.1

Сервіс	Вимоги
API Gateway	CPU: 2-4 ядра, RAM: 4-8 GB, SSD: 20+ GB
Worker Service (Playwright)	CPU: 2-6 ядер, RAM: 4-8 GB, SSD: 20-50 GB
Database Server (PostgreSQL)	CPU: 4 ядра, RAM: 8-16 GB, SSD: 50+ GB
Message Broker (RabbitMQ)	CPU: 2-4 ядра, RAM: 4-8 GB, SSD: < 10 GB
Storage (Azure Blob Storage)	не потребує локальних ресурсів (хмарний сервіс)

Для запуску розробленої системи генерації скріншотів необхідно виконати наступні основні кроки (Таблиця 3.2):

Таблиця 3.2

Етап	Компонент	Опис	Команда
1	Підготовка середовища	Встановлення базових залежностей для запуску системи, включаючи .NET та Docker	Встановити .NET SDK 10.0+ та Docker
2	Розгортання RabbitMQ (Message Broker)	Налаштування та запуск брокера повідомлень, який забезпечує асинхронну взаємодію між сервісами. Конфігурація включає автоматичне створення exchange, черг для різних тарифних планів (Regular, Pro, Advanced) та маршрутизаційних правил.	Відкрити дерикторію WebstoreScreenshotService/Queue та виконати команду: docker compose up -d
3	Розгортання API Gateway	Публікація серверної частини системи та налаштування підключень до зовнішніх сервісів. Після деплою виконується конфігурація доступу до PostgreSQL та RabbitMQ через connection string та environment variables	dotnet publish -c Release -o publish
4	Розгортання Worker Service	Запуск сервісу генерації скріншотів у Docker-контейнері з використанням Playwright та Chromium. Контейнер працює з обмеженнями ресурсів для стабільної обробки браузерних задач	Виконати наступні команди послідовно: docker build -t screenshot-worker-image . docker run -d --name screenshot-service --memory=3g --cpus=4 --shm-size=2g screenshot-worker-image

Розроблена система для взаємодії з користувачем підтримує використання будь-якого сучасного веб-браузера, включаючи Google Chrome (версія

45 і вище), Mozilla Firefox (версія 38 і вище), Microsoft Edge, Safari та інші популярні браузері.

3.2 Структура інтерфейсу платформи

Після першого запуску системи необхідно створити початкового адміністратора платформи. Для цього в конфігураційних налаштуваннях бекенду задається електронна адреса адміністратора, на яку автоматично надсилається запрошення для створення облікового запису (Рис. 3.2).

У листі міститься спеціальне одноразове посилання, перейшовши за яким користувач отримує доступ до форми реєстрації адміністратора системи.

Термін дії токена обмежений та становить 20 хвилин.

Вітаємо,
Для платформи Screenshot Service було створено початковий запит на реєстрацію адміністратора.
Щоб завершити налаштування облікового запису адміністратора, скористайтеся захищеним посиланням нижче:
http://localhost:3000/first-register?token=adm_init_v1_5XQmR9X8QJ2YxYpLwK7mVbN3cDfTzUaH

Це реєстраційне посилання:
може бути використане лише один раз
дійсне протягом 30 хвилин
не підлягає передаванню іншим особам

Screenshot Service

Рис. 3.2. - Лист із запрошенням для створення першого адміністратора

Перейшовши за отриманим посиланням, користувач потрапляє на сторінку створення адміністратора системи (Рис 3.3). На даній сторінці необхідно вказати нікнейм, пароль та підтвердження пароля. Форма містить валідацію введених даних, зокрема перевірку мінімальної довжини пароля та співпадіння паролів.

Рис. 3.3. - Форма створення першого адміністратора системи

Після введення коректних даних користувач може створити адміністратора платформи, натиснувши кнопку «Створіть обліковий запис адміністратора». У випадку успішного створення облікового запису, токен одноразового використання стає недійсним, що унеможлиблює повторну реєстрацію адміністратора за тим самим посиланням. Після успішного створення першого адміністратора системи користувач має можливість здійснити вхід до адміністративної панелі.

На сторінці входу користувачу необхідно ввести нікнейм та пароль, після чого виконати процедуру автентифікації шляхом натискання кнопки «Продовжити». Інтерфейс сторінки виконано у вигляді простої форми авторизації з мінімально необхідними полями для входу в систему.

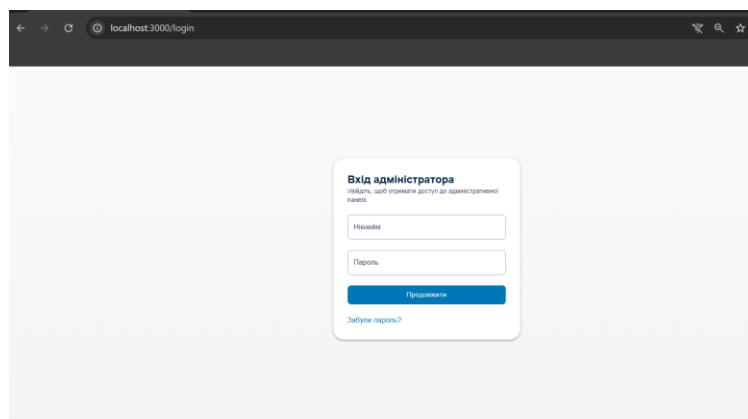


Рис. 3.4. - Форма входу адміністратора в систему

У випадку, якщо користувач забув пароль, йому надається можливість перейти до функціоналу відновлення доступу за допомогою відповідного посилання «Забули пароль?». Однією з функціональних можливостей адміністративної панелі це аналітика доходів. Даний розділ дозволяє адміністратору отримувати інформацію щодо доходів та активності користувачів у системі. На сторінці аналітики (Рис. 3.5) розміщено панель фільтрів, яка дозволяє обирати часовий діапазон (фіксовані періоди або користувацький інтервал дат) та тарифний план для аналізу. Система автоматично перераховує ключові показники, зокрема MRR, прибуток від замовлень, загальний прибуток та кількість користувачів. Нижче відображається графічна візуалізація даних у вигляді кругових діаграм, що демонструють розподіл

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		67

доходів між тарифними планами. Також на сторінці присутня таблиця з деталізацією показників по кожному тарифу.

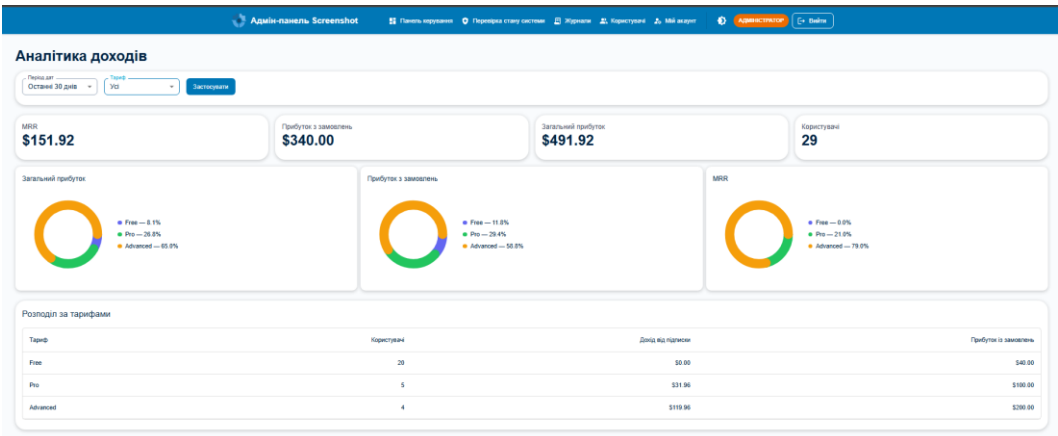


Рис. 3.5. - Сторінка аналітики доходів

Далі розглянемо наступну функцію адміністративної панелі - модуль перевірки стану системи, яка забезпечує моніторинг працездатності всіх ключових компонентів платформи, а також керування режимом обслуговування (Рис. 3.6). Адміністратор має можливість активувати або деактивувати режим обслуговування за допомогою відповідного перемикача, а також виконати перевірку стану системи (health check), який відображає ієрархічне дерево сервісів із їхнім поточним статусом (healthy, degraded, down).

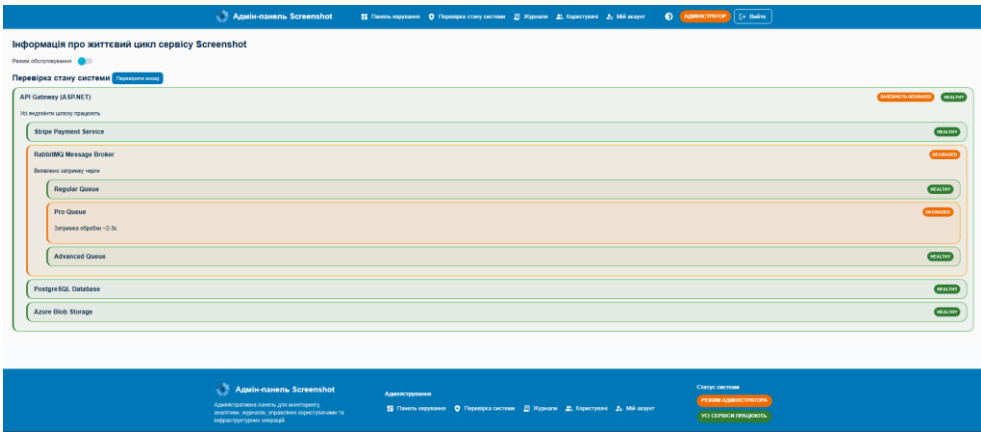


Рис. 3.6. - Сторінка перевірки стану системи та конфігурації режиму обслуговування

При зміні стану режиму обслуговування система додатково відображає модальне вікно (Рис. 3.7) підтвердження, яке запобігає випадковому ввімкненню або вимкненню maintenance mode.

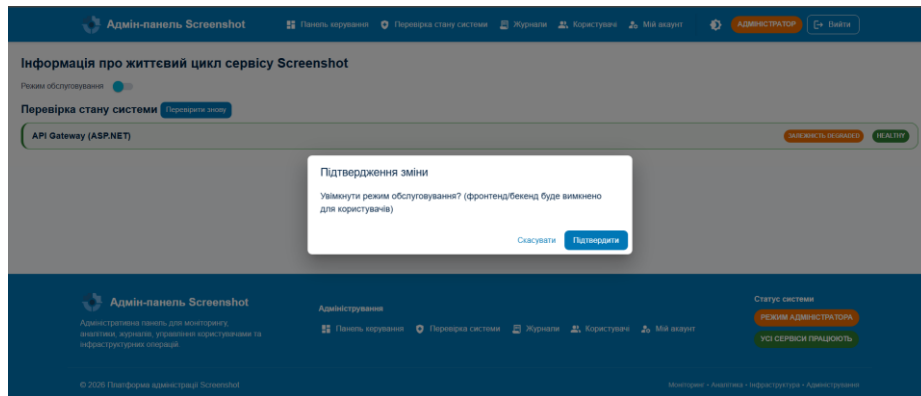


Рис. 3.7. - Підтвердження зміни режиму обслуговування

У випадку активації режиму обслуговування для всіх звичайних користувачів система стає недоступною, а замість основного інтерфейсу відображається спеціальна інформаційна сторінка про технічне обслуговування сервісу(Рис 3.8).

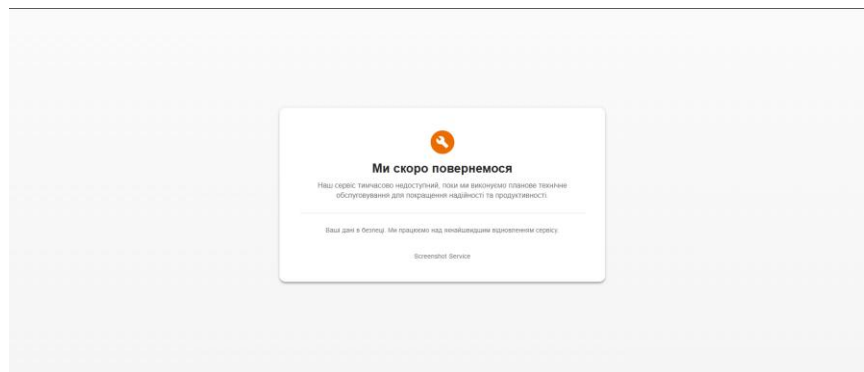


Рис. 3.8. - Сторінка режиму технічного обслуговування (Maintenance Mode Page)

Наступною функціональною можливістю адміністративної панелі є модуль моніторингу логів (Рис. 3.9), що забезпечує централізований перегляд логів. Кожен лог містить інформацію про повідомлення, рівень критичності, джерело, дату створення та додаткові властивості, такі як traceId, середовище виконання та ідентифікатор машини.

У інтерфейсі передбачено можливість фільтрації логів за рівнем критичності, полем пошуку та додатковими атрибутами, включаючи кастомні поля. Також реалізовано пагінацію для зручного перегляду великої кількості записів. Для кожного лог-запису доступне розгортання детальної інформації,

де у вигляді структури JSON відображаються всі властивості події, включаючи технічні параметри виконання.

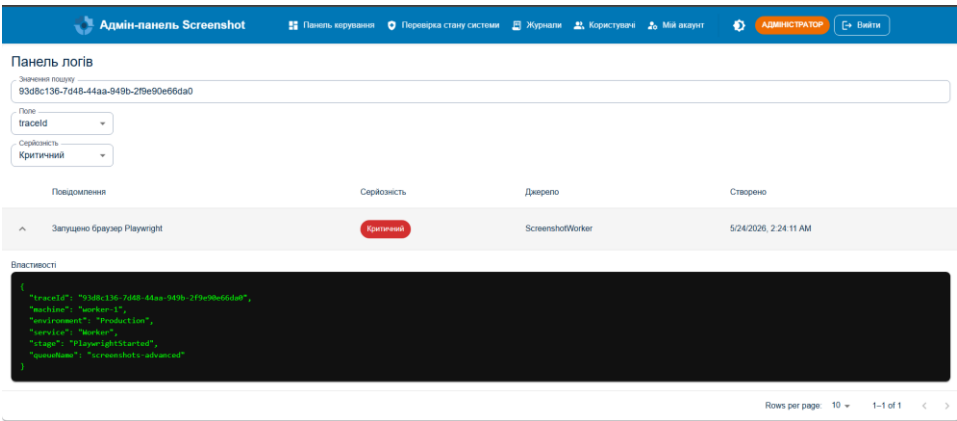


Рис. 3.9. - Панель моніторингу логів системи

Однією з функціональних можливостей адміністративної панелі є модуль керування користувачами (Рис 3.10), який призначений для перегляду, пошуку та адміністрування облікових записів платформи. Даний розділ дозволяє адміністраторам здійснювати фільтрацію користувачів за різними критеріями, зокрема за статусом активності, рівнем підписки та кількістю транзакцій або пошукова панель, яка дозволяє знаходити користувачів за електронною поштою або ім'ям.

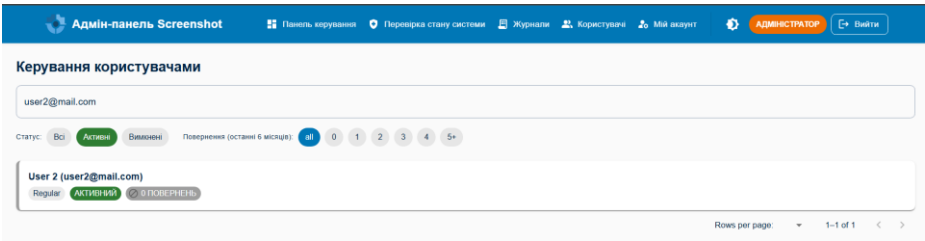


Рис. 3.10. - Пошук та фільтрація користувачів у системі

У цій панелі адміністратор має можливість змінити стан користувача, зокрема активувати або деактивувати його обліковий запис за допомогою відповідної кнопки керування (Рис 3.11).

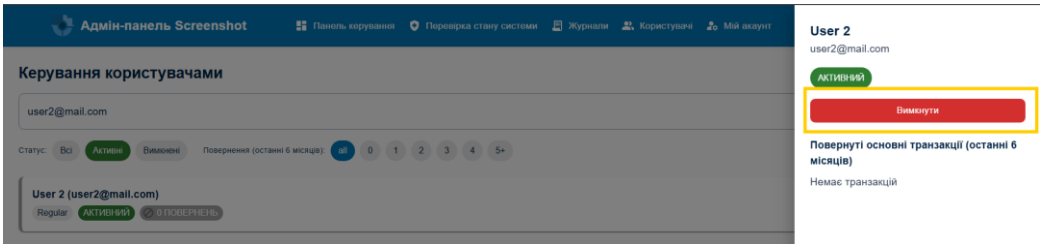


Рис. 3.11. - Панель керування користувачем та зміна статусу акаунта

Останньою сторінкою адміністративної панелі є розділ «Мій акаунт адміністратора» (Рис 3.12), який призначений для перегляду персональної інформації адміністратора системи.

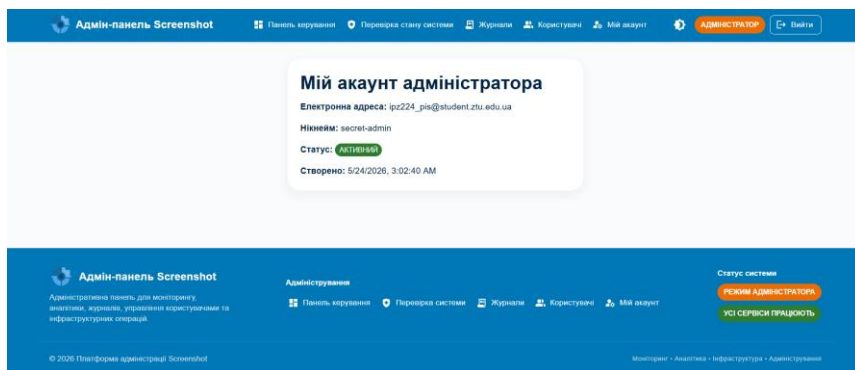


Рис. 3.12. - Сторінка профілю адміністратора (Мій акаунт)

Для початку роботи з платформою користувачу необхідно пройти процедуру реєстрації. Після переходу на сторінку реєстрації він повинен заповнити форму, у якій потрібно вказати електронну пошту, пароль та нікнейм, а також погодитись з політикою конфіденційності (рис. 3.13). Форма містить валідацію введених даних, зокрема перевірку коректності електронної пошти, мінімальної довжини пароля та заповнення обов'язкових полів. Після успішного заповнення форми користувач може створити обліковий запис, натиснувши кнопку «Зареєструватися». Надалі зареєстрований користувач отримує можливість авторизуватись у системі та використовувати функціональність платформи.

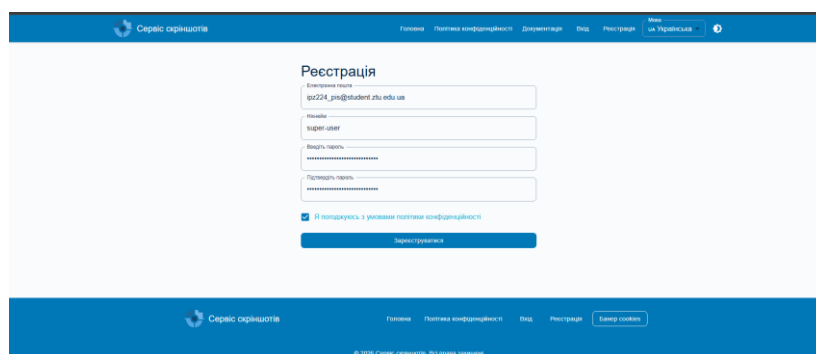


Рис. 3.13. - Форма реєстрації користувача

Після успішної реєстрації користувач отримує можливість авторизуватись у системі за допомогою форми входу (рис. 3.14). Для входу необхідно вказати нікнейм та пароль, які були використані під час створення облікового

запису. Також користувач має можливість перейти до процедури відновлення пароля, натиснувши посилання «Забули пароль?».

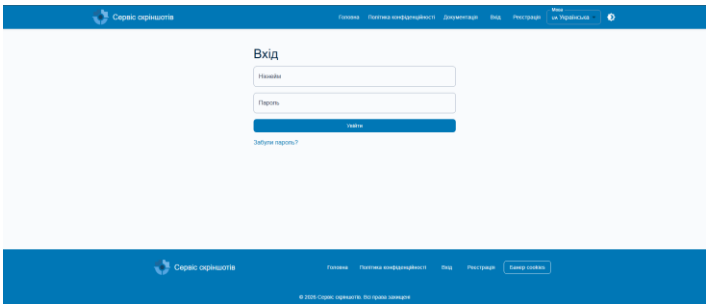


Рис. 3.14. - Форма авторизації користувача

Основний функціонал, який система надає користувачеві - створення веб-скріншотів із гнучким налаштуванням параметрів рендерингу сторінки. Користувач взаємодіє з інтерфейсом, у якому формується конфігурація майбутнього скріншота, включаючи URL сторінки, тип зображення, режим якості, розмір viewport та додаткові параметри емуляції браузера (Рис 3.15)

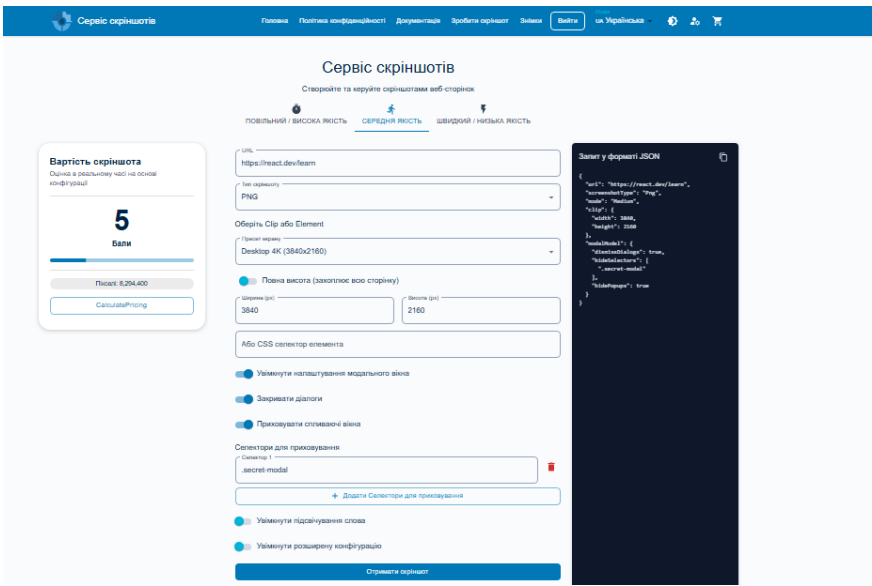


Рис 3.15. - Загальний вигляд сторінки конфігурації

У процесі налаштування користувач обирає режим якості генерації скріншота (Рис. 3.16), який впливає на швидкість виконання запиту та рівень деталізації кінцевого зображення. Передбачено три режими: швидкий, середній та високоякісний. Перемикання між режимами відображається в інтерфейсі у вигляді вкладок або перемикачів, що дозволяє швидко змінювати параметри генерації відповідно до потреб користувача.

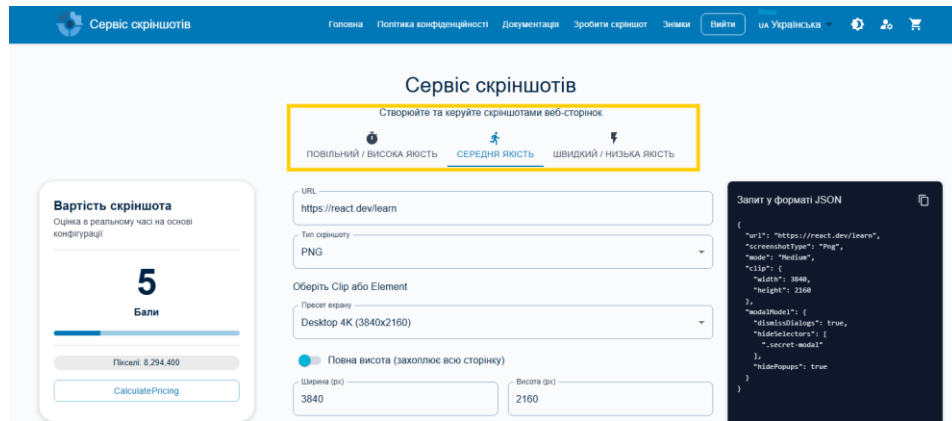


Рис 3.16. - Вибір режиму роботи

Додатково користувач може налаштовувати розширені параметри рендерингу, що дозволяють емуляцію реального браузерного середовища (3.17). До таких параметрів належать локаль, часовий пояс, кольорова схема сторінки, HTTP-заголовки, cookies, а також блокування окремих типів ресурсів. Це дозволяє створювати скріншоти у максимально наближених до реальних умов середовища сценаріях.

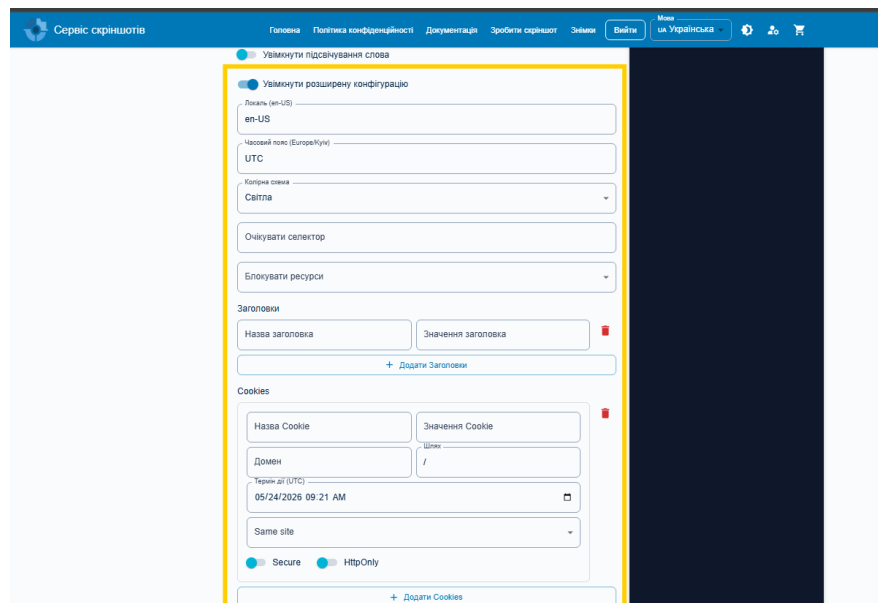


Рис 3.17. - Інтерфейс розширених налаштувань

Після виконання запиту користувач отримує готовий результат у вигляді зображення, яке може бути переглянуте безпосередньо в інтерфейсі, відкрито у новій вкладці або завантажено на локальний пристрій (Рис. 3.18). Також система відображає прев'ю згенерованого скріншота разом із JSON-запитом, який зручно використовувати під час інтеграції через API, ко-

ристувач може налаштувати параметри генерації знімків безпосередньо на вебсайті, а потім застосувати ці ж налаштування для роботи з API.

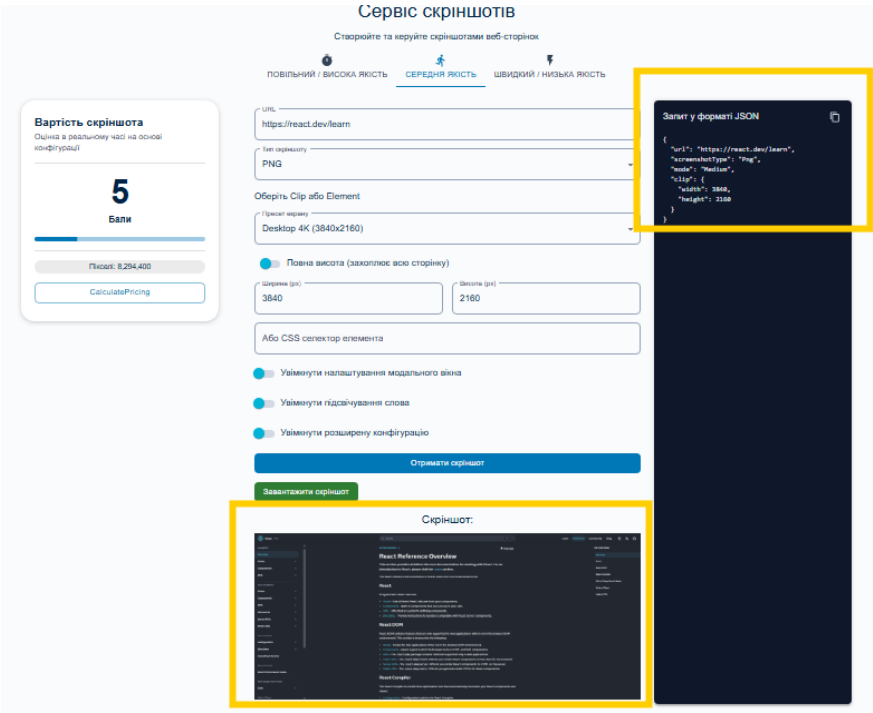


Рис 3.18. - Приклад сторінки результату.

Наступним розглянемо функціональність перегляду збережених скріншотів. На сторінці списку (рис. 3.19) відображається список знімків із базовими інформацією, включаючи URL, тип, категорії, статус обробки та дату створення. Для зручності пошуку передбачено фільтри за текстовим запитом, категоріями та параметрами вибірки, а також реалізовано пагінацію для роботи з великим обсягом даних.

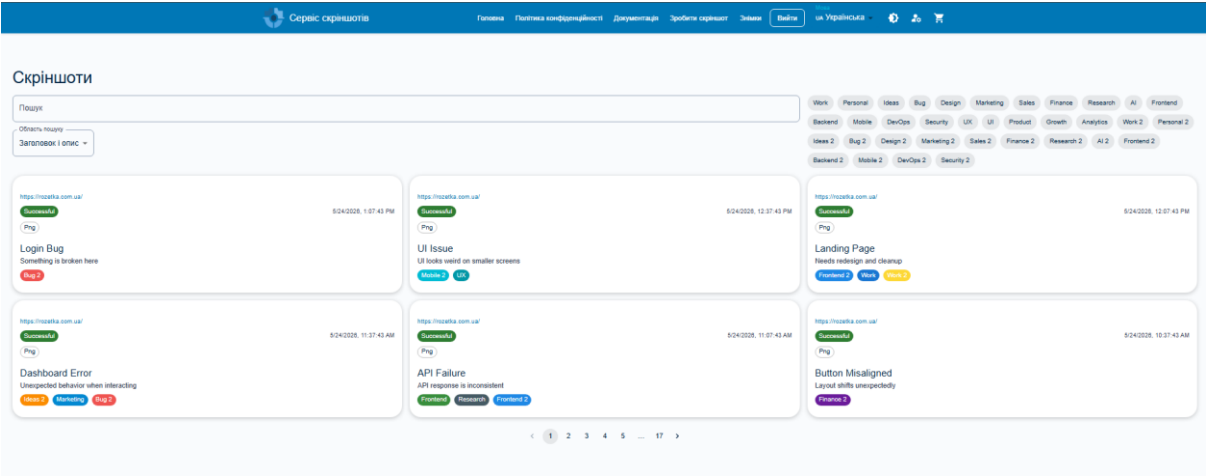


Рис 3.19. - Сторінка списку збережених скріншотів

Сторінка детального перегляду (рис. 3.20) відображає сам знімок та супровідну інформацію: назву, опис, URL вебресурсу, статус обробки, тип скріншота, категорії та дату створення. Залежно від стану обробки може відображатися завантаження, успішний результат або повідомлення про помилку генерації.

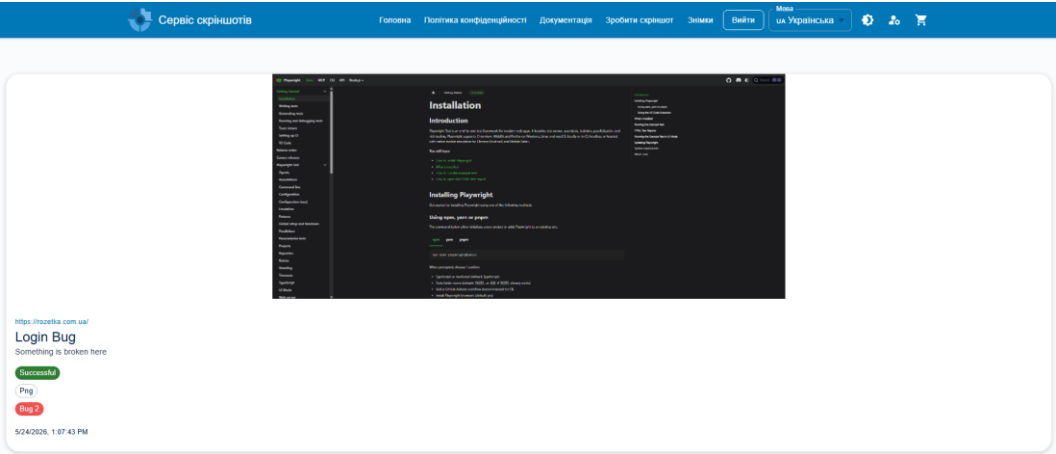


Рис 3.20. - Сторінка деталей збереженого скріншоту

Функціонал покупки балів та підписки є одним із ключових бізнес-модулів сервісу, оскільки саме він забезпечує монетизацію. Він реалізований через окрему секцію на головній сторінці (Рис 3.21), де користувач може ознайомитися з тарифними планами та обрати потрібний варіант підписки або поповнення балансу.

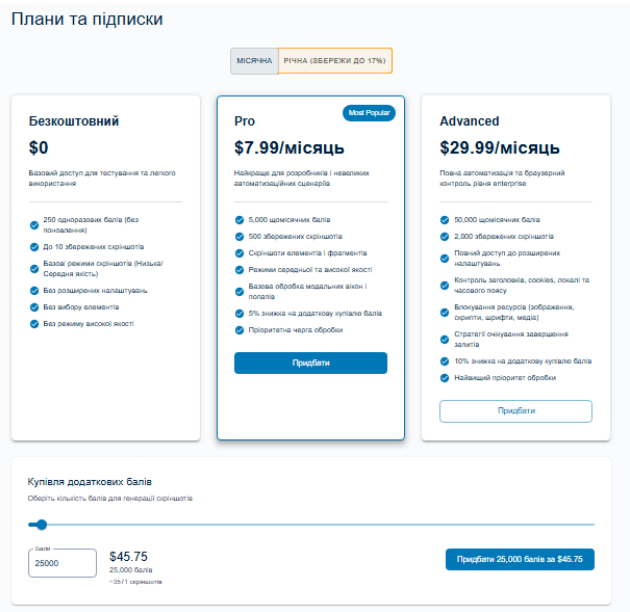


Рис 3.21. - Секція покупки балів та підписок

На головній сторінці (рис. 3.22) відображено блок підписок, який містить декілька тарифних планів. Користувач може перемикати період оплати (місячний або річний), після чого автоматично перераховується ціна, також, для кожного плану наведено перелік переваг.

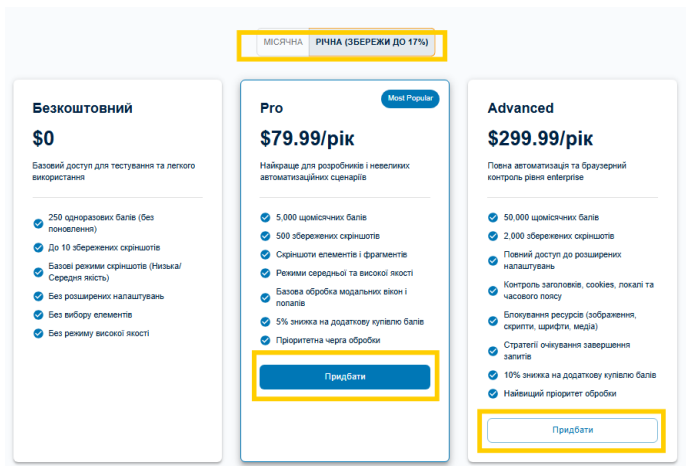


Рис 3.22. - Секція покупки підписок

Після вибору тарифу відкривається модальне вікно підтвердження (Рис. 3.23), у якому користувач бачить обраний план, його вартість та перелік можливостей, та має можливість придбати вибраний план завдяки інтегрованому платіжному компоненту (Stripe checkout).

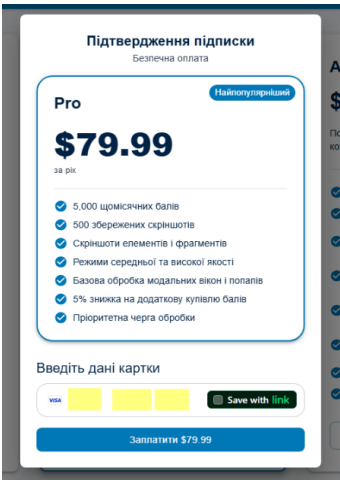


Рис 3.23. - Модальне вікно підтвердження покупки підписки

Окремо реалізований функціонал купівлі додаткових балів (Рис 3.24), який дозволяє користувачу поповнювати баланс для генерації скріншотів. Інтерфейс містить слайдер вибору кількості балів, поле ручного введення, ав-

томатичний розрахунок вартості та приблизної кількості доступних скріншотів.

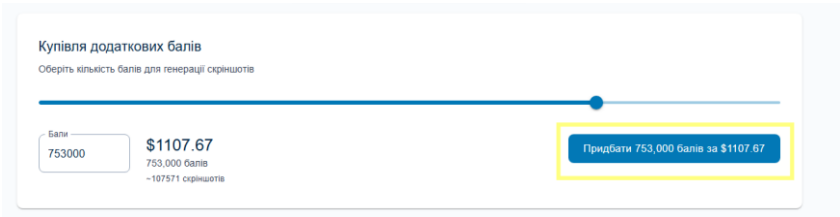


Рис 3.24. - Функціональність купівлі додаткових балів

Після вибору пакету балів або підписки користувач не одразу здійснює фінальну оплату, а спочатку додає обраний товар до кошика (Рис. 3.25). Це дозволяє сформувати замовлення та переглянути його перед підтвердженням, що є типовим підходом для e-commerce сценаріїв у сервісах підписки.

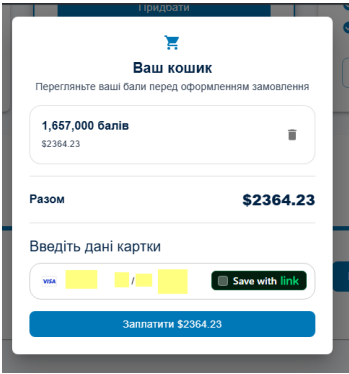


Рис. 3.25. - Кошик

Далі розглянемо можливості, які відносяться до особистого кабінету користувача. Перша з яких - «Мій акаунт» (Рис. 3.26) яка призначена для перегляду основної інформації про профіль, активну підписку та доступний баланс балів. Крім цього, користувач має можливість перейти до сторінки зміни тарифного плану, натиснувши кнопку «Змінити план»

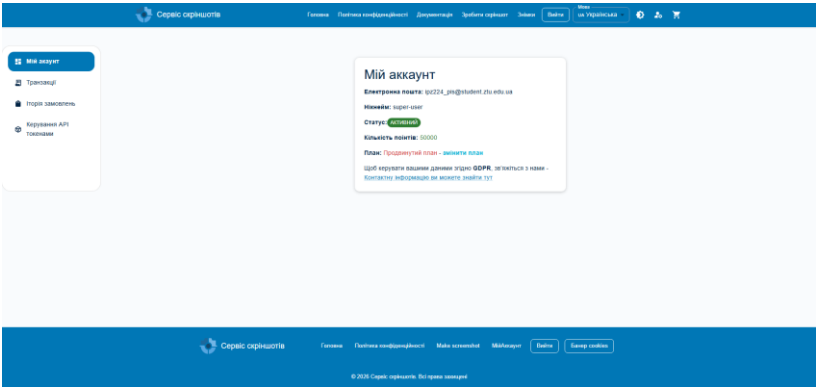


Рис. 3.26. - Функціональність «Мій акаунт»

Для користувачів, які мають активну платну підписку, система також надає можливість її скасування. Натиснувши кнопку «Скасувати підписку», користувачеві відображається модальне вікно підтвердження (Рис. 3.27), у якому система повідомляє про умови скасування підписки.

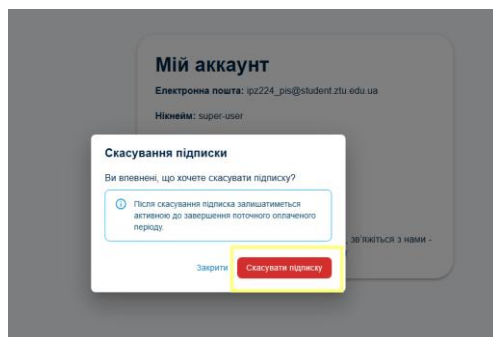


Рис. 3.27. - Підтвердження скасування підписки

Після підтвердження скасування підписки, користувачеві більше не буде автоматично продовжено тарифний план, однак підписка залишатиметься активною до завершення поточного платіжного періоду (зазвичай до кінця місяця або року).

Функціонал керування транзакціями реалізовано як окремий модуль кабінету користувача (Рис. 3.28), який забезпечує перегляд фінансових транзакцій та можливість ініціювати повернення коштів.

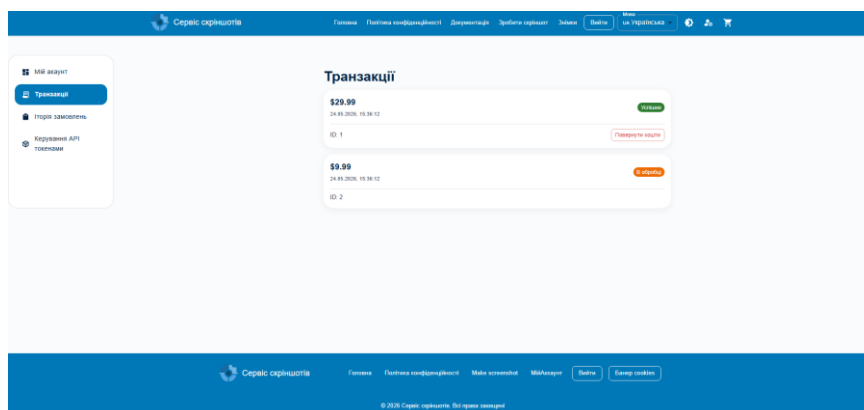


Рис. 3.28 Функціонал керування транзакціями

У разі успішності транзакції користувачу надається можливість ініціювати повернення коштів (3.29). При натисканні відповідної кнопки відкривається модальне вікно підтвердження, яке запобігає випадковому створенню запиту. Після підтвердження викликається кошти повертаються користувачу. Для запобігання зловживання, користувач має право повернути до

3 транзакцій у період 6 місяців, якщо буде спроба повернути більше, то акаунт буде заблоковано.

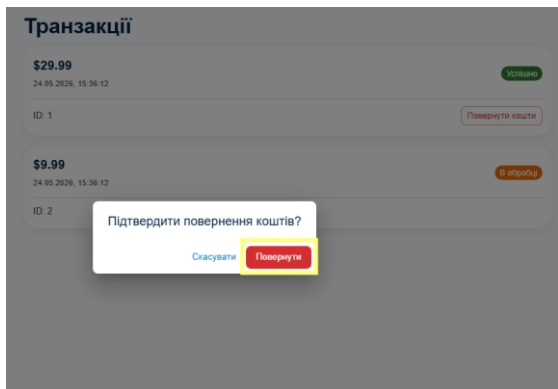


Рис. 3.29. - Модал підтвердження повернення коштів

Функціонал перегляду замовлень реалізовано як частину особистого кабінету користувача та забезпечує доступ до історії всіх оформлених замовлень. На сторінці списку замовлень (Рис 3.30) користувачу відображається список, який містить ідентифікатор замовлення, поточний статус, загальну суму та дату створення. Для зручності користувач може здійснювати пошук за номером замовлення, відфільтровувати записи за статусом (наприклад, оплачено, очікує оплати або скасовано), а також змінювати порядок відображення залежно від дати.

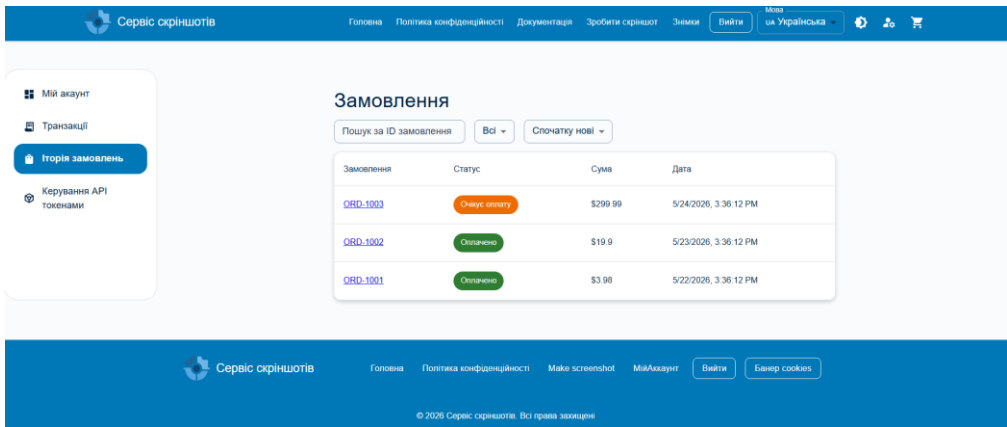


Рис. 3.30. - Сторінка списку замовлень

Окремо реалізовано сторінку деталей замовлення (Рис 3.31), яка відкривається при виборі конкретного запису. На цій сторінці користувач може переглянути повну інформацію про замовлення. Додатково відображається перелік позицій замовлення, де для кожного товару показано назву, кількість, ціну за одиницю та вартість.

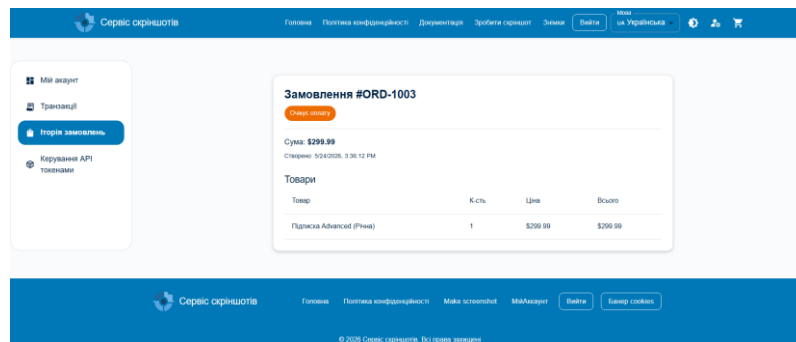


Рис. 3.31. - Сторінка детального перегляду замовлення

Останньою, проте не менш важливою функціональністю платформи є керування API токенами що дозволяє користувачам створювати, переглядати та відкликати токени доступу, які надалі можуть використовуватись для інтеграції із зовнішніми системами та автоматизації взаємодії через API сервісу. Для перегляду наявних токенів користувач переходить на сторінку керування API токенами (Рис. 3.32), де відображається список усіх створених токенів. Для кожного токена відображається назва, поточний статус, дозволи, дозволені IP-адреси, а також метадані щодо створення, останнього використання та відкликання токена.

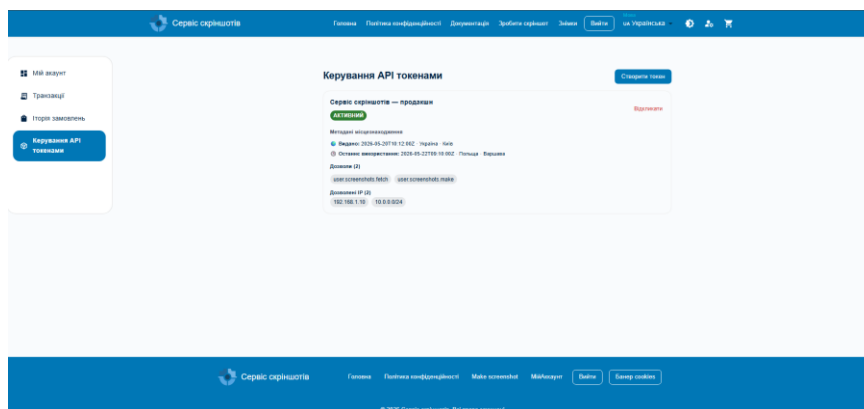


Рис. 3.32. - Перегляд та керування API токенами

Натиснувши кнопку «Створити токен», користувачеві відображається форма створення нового API токена (Рис. 3.33) у якій користувач може вказати назву токена, обрати необхідні сфери доступу, а також налаштувати список дозволених IP-адрес або CIDR-діапазонів, з яких дозволитиметься використання токена.

Рис. 3.33. - Форма створення API токена

Після введення коректних даних та натискання кнопки «Створити» система генерує новий API токен та відображає його користувачеві лише один раз (Рис. 3.34).

Рис. 3.34. - Відображення згенерованого API токена

У випадку необхідності відкликання певного токена користувач може натиснути кнопку «Відкликати» на сторінці керування токенами (Рис. 3.32). Після цього користувачу відображається форма підтвердження відкликання токена (Рис. 3.35), у якій можна вказати причину деактивації.

Рис. 3.35. - Відкликання API токена

Після відкликання токен змінює свій статус на неактивний, а інформація про час та місце відкликання відображається у метаданих токена (Рис. 3.36). Це дозволяє здійснювати аудит використання API токенів та контролювати доступ до системи.

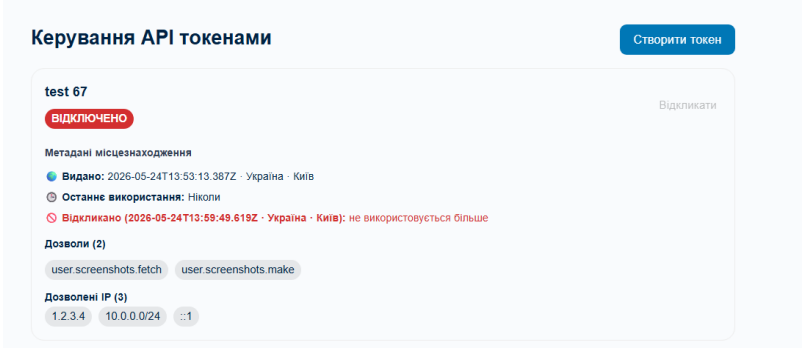


Рис. 3.36. - Відображення відкликаного API токена

Також після створення API токена користувач має можливість переглянути документацію API сервісу на окремій сторінці (Рис. 3.36) на якій відображається перелік доступних endpoint, методи HTTP-запитів, маршрути, моделі запитів та відповідей, приклади використання, а також можливі коди відповіді сервера.

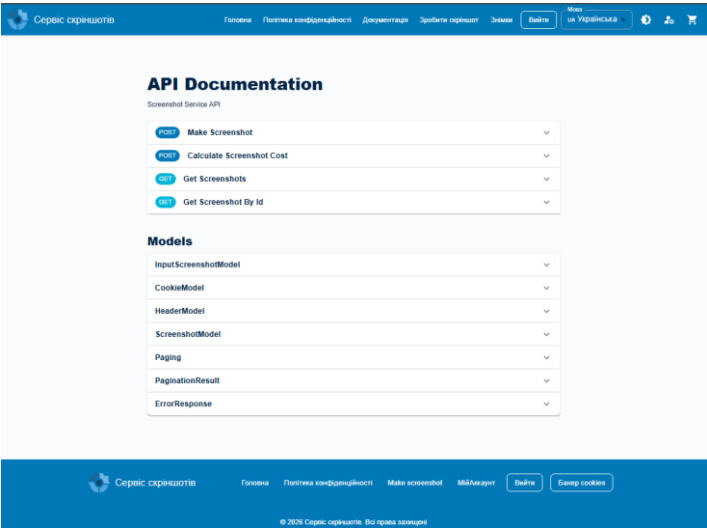


Рис. 3.37. - Документація API платформи

3.3 Тестування програмного системи

У процесі розробки платформи для генерації скріншотів було використано комбінований підхід до тестування програмного забезпечення, який включає ручне тестування користувацького інтерфейсу та тестування API на

рівні HTTP-запитів. Основним методом перевірки коректності роботи веб-інтерфейсу було ручне тестування. Ручне тестування застосовувалося для всіх основних функціональних модулів системи, включаючи реєстрацію користувачів, створення скріншотів, керування підписками та роботу адміністративної панелі.

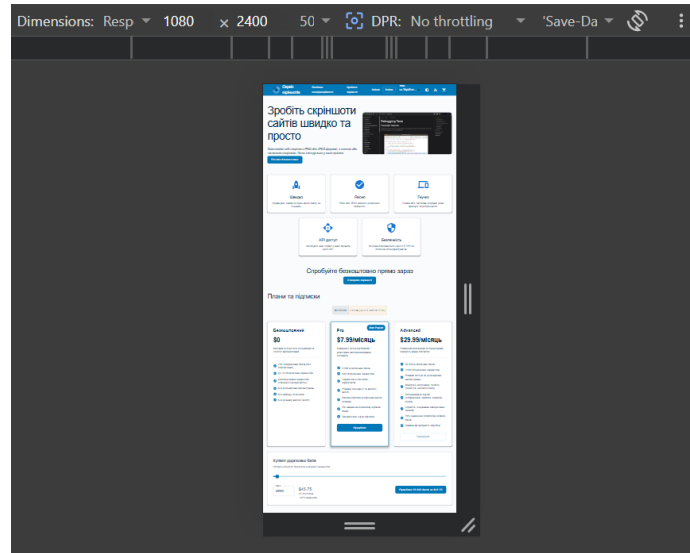


Рис. 3.38. - Ручне тестування інтерфейсу головної сторінки користувача

Окрему частину тестування становила перевірка API-інтерфейсу системи. Для цього використовувався інструмент командного рядка curl, який дозволяє виконувати HTTP-запити до серверної частини та перевіряти коректність відповідей API. Такий підхід забезпечив можливість швидкої перевірки окремих endpoint-ів без необхідності використання графічних клієнтів.

```
\WebstoreScreenshotService\ScreenshotStorage>curl -X GET "http://localhost:5172/getScreenshot?id=8f3c2a1d-6c5b-4d91-9c2a-7f1e4a8d3b60" -H "Authorization: api_v1.eys3d4e5" -H "Content-Type: application/json"
```

```
{
  "Id": "8f3c2a1d-6c5b-4d91-9c2a-7f1e4a8d3b60",
  "WebsiteUrl": "https://react.dev/reference/react",
  "Url": "http://localhost:4172/55973cca-2aa9-4d50-bbab-cb015c05cda8/8f3c2a1d-6c5b-4d91-9c2a-7f1e4a8d3b60.png",
  "CreatedAt": "2026-05-24T17:42:34.6468226Z",
  "State": "Completed",
  "Type": "Png"
}
```

Рис. 3.39. - Тестування screenshotGet API запиту за допомогою curl

На даному етапі розвитку системи тестування API виконується переважно вручну, однак у майбутньому планується впровадження інструментів типу Postman або аналогічних рішень для автоматизації та структурованого тестування великої кількості endpoint-ів. Це дозволить створювати колекції

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ІПЗ	Арк.
		Єфремов Ю.М.				83
Змн.	Арк.	№ докум.	Підпис	Дата		

запитів, автоматизувати регресійне тестування та підвищити загальну якість контролю API.

Таким чином, використаний підхід до тестування забезпечує достатній рівень контролю якості на поточному етапі розробки системи та створює основу для подальшої автоматизації тестових процесів.

Висновки до третього розділу

У третьому розділі було розглянуто процес розгортання та налаштування масштабованого сервісу для генерації скріншотів. Зокрема, було описано загальну архітектуру системи, що базується на взаємодії незалежних сервісів, таких як API Gateway, Worker Service, брокер повідомлень RabbitMQ, база даних PostgreSQL та хмарне сховище Blob Storage. Також було наведено характеристики серверної інфраструктури та вимоги до ресурсів для кожного з компонентів системи.

Окрему увагу приділено процесу розгортання основних компонентів платформи, включаючи налаштування середовища виконання, запуск брокера повідомлень, публікацію API Gateway та контейнеризацію Worker Service, що виконує генерацію скріншотів за допомогою Playwright.

У розділі також детально описано інтерфейс користувацької та адміністративної частини системи, включаючи основні функціональні модулі, такі як створення скріншотів, керування підписками, аналітика, моніторинг системи, робота з транзакціями та API токенами.

У підрозділі тестування було наведено підхід до перевірки працездатності системи, який включає ручне тестування користувацького інтерфейсу та тестування API за допомогою інструменту curl. Також зазначено перспективу подальшого впровадження інструментів автоматизованого тестування, зокрема Postman, для розширення можливостей перевірки API.

У результаті було забезпечено перевірку коректності роботи основних функціональних модулів системи та підтверджено працездатність реалізованої мікросервісної архітектури в умовах моделювання реального навантаження.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		84

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проаналізовано предметну область та існуючі рішення для генерації скріншотів вебсторінок, що дозволило визначити ключові вимоги до майбутньої системи. Сформовано функціональні та нефункціональні вимоги, зокрема щодо масштабованості, продуктивності, безпеки та зручності використання.

У першому розділі проведено огляд аналогічних сервісів, визначено їхні недоліки та обґрунтовано вибір мікросервісної архітектури, яка забезпечує гнучкість, незалежне масштабування та стійкість до відмов. Обрано технологічний стек: C#, ASP.NET Core, PostgreSQL, RabbitMQ, а також React із Redux та RxJS для клієнтської частини.

Другий розділ присвячено проєктуванню сервісу: визначено основні сутності та їх взаємозв'язки, побудовано доменну модель, описано структуру мікросервісів. Особливу увагу приділено асинхронній обробці задач на основі RabbitMQ, що забезпечує ефективний розподіл навантаження та виконання ресурсоемних операцій. Описано механізми взаємодії сервісів і доступу до даних.

У третьому розділі розглянуто процес розгортання та тестування сервісу. Наведено діаграму розгортання, описано порядок встановлення та налаштування компонентів: API Gateway, Worker Service, RabbitMQ, PostgreSQL та сховища даних. Визначено мінімальні вимоги до інфраструктури та описано етапи запуску.

У результаті створено повноцінний масштабований сервіс для генерації скріншотів вебсторінок із серверною, клієнтською та інфраструктурною частинами. Система відповідає принципам масштабованості, модульності та розширюваності, що робить її готовою до практичного використання та подальшого розвитку.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		85

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Назаренко О.М., Кравченко І.В. Аналіз сучасних підходів до побудови мікросервісної архітектури вебзастосунків. Вісник Хмельницького національного університету. Технічні науки. 2023. № 2. С. 112-118.
2. PostgreSQL Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://www.postgresql.org/docs/>.
3. ASP.NET Core documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/aspnet/core/>.
4. Entity Framework Core Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/ef/core/>.
5. RabbitMQ Official Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://www.rabbitmq.com/documentation.html>.
6. Introduction to RabbitMQ [Електронний ресурс]. - Режим доступу до ресурсу: <https://jstobigdata.com/rabbitmq/introduction-to-rabbitmq/>.
7. Microservice Architecture pattern [Електронний ресурс]. - Режим доступу до ресурсу: <https://microservices.io/patterns/microservices.html>.
8. Що краще моноліт чи мікросервіси? | IAMPM [Електронний ресурс]. - 2023. - Режим доступу до ресурсу: <https://iampm.club/ua/blog/shho-krashhe-monolit-chi-mikroservisi-yak-obrati-arhitekturu-projektu/>.
9. Про мікросервісну архітектуру [Електронний ресурс]. - 2024. - Режим доступу до ресурсу: <https://foxminded.ua/mikroservisna-arkhitektura/>.
10. Навіщо потрібна сервіс-орієнтована архітектура | robot_dreams [Електронний ресурс]. - Режим доступу до ресурсу: <https://robotdreams.cc/uk/blog/268-zachem-nuzhna-servis-orientirovannaya-arhitektura>.
11. What is the Role of API Gateway in Microservices? [Електронний ресурс]. - 2024. - Режим доступу до ресурсу: <https://www.geeksforgeeks.org/what-is-the-role-of-api-gateway-in-microservices/>.
12. Azure Blob Storage documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/azure/storage/blobs/>.

- 13.Playwright Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://playwright.dev/docs/intro>.
- 14.gRPC Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://grpc.io/docs/>.
- 15.Docker Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://docs.docker.com/>.
- 16.Kubernetes Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://kubernetes.io/docs/>.
- 17.Clean Architecture by Robert C. Martin [Електронний ресурс]. - Режим доступу до ресурсу: <https://blog.cleancoder.com/>.
- 18.Repository Pattern in ASP.NET Core [Електронний ресурс]. - Режим доступу до ресурсу: <https://codewithmukesh.com/blog/repository-pattern-in-aspnet-core/>.
- 19.Result Pattern in .NET [Електронний ресурс]. - Режим доступу до ресурсу: <https://dev.to/milanjovanovictch/functional-error-handling-in-net-with-the-result-pattern-1cj1>.
- 20.TypeScript Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://www.typescriptlang.org/docs/>.
- 21.React Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://react.dev/>.
- 22.Redux Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://redux.js.org/>.
- 23.RxJS Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://rxjs.dev/guide/overview>.
- 24.Redux-Observable Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://redux-observable.js.org/>.
- 25.GitLab Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://docs.gitlab.com/>.
26. ASP.NET Web API Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/aspnet/web-api/>.

27. .NET Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/dotnet/>.
28. PostgreSQL Global Development Group. PostgreSQL Documentation (SQL Language Reference) [Електронний ресурс]. - Режим доступу: <https://www.postgresql.org/docs/current/sql.html>
29. PostgreSQL Tutorial [Електронний ресурс]. - Режим доступу до ресурсу: <https://www.postgresqltutorial.com/>.
30. ASP.NET Core Security Documentation [Електронний ресурс]. - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/aspnet/core/security/>.

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		88

ДОДАТКИ

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		89

ЛІСТИНГ ВИХІДНОГО КОДУ

```

[Authorize(Roles = UserRoles.User)]
[ApiController]
[Route("screenshots/[action]")]
public class ScreenshotController(
    IScreenshotService screenshotService,
    IScreenshotManager screenshotManager,
    IScreenshotStorageManager screenshotStorageManager,
    IUserContextAccessor userContextAccessor,
    IScreenshotCalculator screenshotCalculator) : ControllerBase
{
    private readonly IScreenshotService _screenshotService =
        screenshotService;

    private readonly IScreenshotManager _screenshotManager =
        screenshotManager;

    private readonly IScreenshotStorageManager _screenshotStorageManager =
        screenshotStorageManager;

    private readonly IUserContextAccessor _userContextAccessor =
        userContextAccessor;

    private readonly IScreenshotCalculator _screenshotCalculator =
        screenshotCalculator;

    [HttpPost]
    [Authorize(Policy = Policies.User.MakeScreenshots)]
    [ActionName("makeScreenshot")]
    [ProducesResponseType<FileStream>(StatusCodes.Status200OK, "image/png",
        "image/jpeg")]
    [ProducesResponseType<ValidationProblemDetails>(StatusCodes.Status400BadRequest,
        "application/json")]
    [SwaggerResponseExample(StatusCodes.Status400BadRequest,
        typeof(MakeScreenshotResponseExample))]

```

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ІЗ	Арк.
		Єфремов Ю.М.				90
Змн.	Арк.	№ докум.	Підпис	Дата		

```

    public async Task<IActionResult> MakeScreenshot([FromBody]
InputScreenshotModel screenshotOptions)

    {

        var userContext = _userContextAccessor.GetCurrentUser();

        var model = screenshotOptions.ToScreenshotOptions(userContext);

        if (!model.IsSuccess)

            return BadRequest(new ErrorResponse(model.ErrorMessage!));

        var screenshotResult = await
        _screenshotService.MakeScreenshotAsync(model.Value!);

        if (!screenshotResult.IsSuccess)

            return BadRequest(new
ErrorResponse(screenshotResult.ErrorMessage!));

        var useInfo = userContext.UserInfo;

        var screenshot = screenshotResult.Value!;

        var responseModel = new ScreenshotModel(screenshot,
        _screenshotStorageManager.GetScreenshotUrl(screenshot, useInfo.Id));

        return Ok(responseModel);

    }

```

[HttpPost]

[Authorize(Policy = Policies.User.MakeScreenshots)]

[ActionName("calculateScreenshotCost")]

```

    public IActionResult CalculateScreenshotCost([FromBody]
InputScreenshotModel screenshotOptions)

```

```

    {

        var userContext = _userContextAccessor.GetCurrentUser();

        var model = screenshotOptions.ToScreenshotOptions(userContext);

        if (!model.IsSuccess)

            return BadRequest(new ErrorResponse(model.ErrorMessage!));

    }

```

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		91

```

        var pointsCost = _screenshotCalculator.CalculatePoints(model.Value!);
        var calculationModel = new ScreenshotCalculationModel(pointsCost);

        return Ok(calculationModel);
    }

    [HttpGet]
    [Authorize(Policy = Policies.User.FetchScreenshots)]
    [ActionName("getScreenshots")]
    public async Task<IActionResult> GetScreenshots([FromQuery] Paging
    paging)
    {
        var userContext = _userContextAccessor.GetCurrentUser();

        var screenshotPaging = new ScreenshotPaging(paging.Page,
        paging.PageSize, paging.Query, paging.SearchScope, paging.CategoryIds);

        var screenshotResult = await
        _screenshotManager.GetScreenshots(screenshotPaging);

        if (!screenshotResult.IsSuccess)

            return BadRequest(new
            ErrorResponse(screenshotResult.ErrorMessage!));

        var result = screenshotResult.Value!;

        var userInfo = userContext.UserInfo;
        var paginationResult = new PaginationResult<ScreenshotModel>
        (
            TotalCount: result.TotalCount,
            Items: result.Items

                .Select(screenshot => new ScreenshotModel(screenshot,
                _screenshotStorageManager.GetScreenshotUrl(screenshot, userInfo.Id)))

```

```

        .ToArray()

    );

    return Ok(paginationResult);
}

[HttpGet]
[Authorize(Policy = Policies.User.FetchScreenshots)]
[ActionName("getScreenshot")]
public async Task<IActionResult> GetScreenshot([FromQuery] string id)
{
    var userContext = _userContextAccessor.GetCurrentUser();

    var storedScreenshotResult = await
        _screenshotManager.GetScreenshot(id);

    if (!storedScreenshotResult.IsSuccess)

        return BadRequest(new
            ErrorResponse(storedScreenshotResult.ErrorMessage!));

    var useInfo = userContext.UserInfo;

    var storedScreenshot = storedScreenshotResult.Value!;

    var screenshot = new ScreenshotModel(storedScreenshot,
        _screenshotStorageManager.GetScreenshotUrl(storedScreenshot, useInfo.Id));

    return Ok(screenshot);
}

}

public class ScreenshotService(
    IUserContextAccessor userContextAccessor,
    IMessageBrokerManager messageBrokerManager,
    IOptions<MessageBrokerConfigurations> options,

```

```

    IAuthorizationManager authorizationManager,
    IScreenshotManager screenshotManager,
    ISubscriptionManager subscriptionManager,
    IScreenshotCalculator screenshotCalculator) : IScreenshotService
{
    private readonly ISubscriptionManager _subscriptionManager =
subscriptionManager;

    private readonly IUserContextAccessor _userContextAccessor =
userContextAccessor;

    private readonly IMessageBrokerManager _messageBrokerManager =
messageBrokerManager;

    private readonly IAuthorizationManager _authorizationManager =
authorizationManager;

    private readonly IScreenshotManager _screenshotManager =
screenshotManager;

    private readonly IScreenshotCalculator _screenshotCalculator =
screenshotCalculator;

    private readonly QueueConfig _queueConfig = options.Value.Queue;

    private readonly Result<Screenshot> defaultErrorMessage =
Result<Screenshot>.Error("Failed to send screenshot request. Please, try
again later");

    /// <summary>

    /// Takes a screenshot of a webpage based on the specified options.

    /// </summary>

    /// <param name="screenshotOptionsModel">The options for taking the
screenshot.</param>

    /// <returns>A task that represents the asynchronous operation. The task
result contains the screenshot as a stream.</returns>

    public async Task<Result<Screenshot>>
MakeScreenshotAsync(ScreenshotOptionsModel screenshotOptionsModel)
    {
        var userContext = _userContextAccessor.GetCurrentUser();

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		94

```

var useInfo = userContext.UserInfo;

var screenshotPlan = userContext.SubscriptionPlan;

var screenshotId = MakeScreenshotId();

var pointsCost =
_screenshotCalculator.CalculatePoints(screenshotOptionsModel);

var screenshotResult = await
_subscriptionManager.ScreenshotWasMadeAsync(new(pointsCost));

if (!screenshotResult.IsSuccess)

    return Result<Screenshot>.Error(screenshotResult.ErrorMessage!);

var confirmationToken =
_authorizationManager.GenerateConfirmationToken(new ConfirmationData
(
    UserId: useInfo.Id,
    ScreenshotId: screenshotId,
    PointsCost: pointsCost,
    TokenId: Guid.CreateVersion7().ToString("N")
));

if (confirmationToken is null)

    return defaultErrorMessage;

var savedScreenshotResult = await _screenshotManager.MakeAsync(new
(
    Id: screenshotId,
    UserId: useInfo.Id,
    WebsiteUrl: screenshotOptionsModel.Url,
    Type: screenshotOptionsModel.ScreenshotType,

```

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ІЗ	Арк.
		Єфремов Ю.М.				95
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        PointsCost: pointsCost

    ));

    if (!savedScreenshotResult.IsSuccess)

        return
        Result<Screenshot>.Error(savedScreenshotResult.ErrorMessage!);

    var savedScreenshot = savedScreenshotResult.Value!;

    var model = new MakeScreenshotModel
    {
        ScreenshotId = savedScreenshot.Id,
        ScreenshotOptionsModel = screenshotOptionsModel,
        ConfirmationToken = confirmationToken,
        UserInformation = new UserInformation
        {
            UserId = useInfo.Id,
        }
    };

    var routingKey =
    _queueConfig.QueuePerSubscription[screenshotPlan.Type];

    var successfullySent = await
    _messageBrokerManager.SendMessageAsync(model, routingKey);

    if (!successfullySent)
    {
        await
        _subscriptionManager.RedeemScreenshotAsync(new(savedScreenshot.Id,
        pointsCost));
    }

```

```

        return defaultErrorMessage;
    }

    return Result<Screenshot>.Success(savedScreenshot);
}

private static string MakeScreenshotId()
    => Guid.CreateVersion7().ToString();
}

public class UserContextAccessor(IHttpContextAccessor httpContextAccessor) :
    IUserContextAccessor
{
    private readonly IHttpContextAccessor _httpContextAccessor =
        httpContextAccessor;

    public UserContext? TryCurrentUser()
        => _httpContextAccessor.HttpContext?.GetUserContext();

    public UserContext GetCurrentUser()
        => _httpContextAccessor.HttpContext?.GetUserContext()
            ?? ThrowInvalidOperation<UserContext>();

    public UserSpecificServices? TryUserSpecificServices()
        => _httpContextAccessor.HttpContext?.GetUserSpecificServices();

    public UserSpecificServices GetUserSpecificServices()
        => _httpContextAccessor.HttpContext?.GetUserSpecificServices()
            ?? ThrowInvalidOperation<UserSpecificServices>();

    private static T ThrowInvalidOperation<T>()

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		97

```

=> throw new InvalidOperationException($"{typeof(T)} was not set.
This may indicate an invalid or missing authentication token.");
}

public class ScreenshotPricingCalculator: IScreenshotCalculator
{
    private const decimal BasePoints = 1.0M;

    private const int DefaultWidth = 1920;
    private const int DefaultHeight = 1080;

    public int CalculatePoints(ScreenshotOptionsModel options)
    {
        var pixels = CalculatePixels(options);

        var sizeMultiplier = pixels / (decimal)(DefaultWidth *
DefaultHeight);

        var total = BasePoints * sizeMultiplier;

        // Feature multipliers
        if (IsFullPage(options))
            total *= 1.4M;

        if (options.Element != null)
            total *= 1.5M;

        // Loading complexity
        total *= GetLoadingMultiplier(options.ContentLoadingOptions);

        //advanced settings complexity

```

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ІПЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		98

```

        total += GetAdvancedCost(options.AdvancedConfiguration);

// Additive costs
if (options.HighlightWord != null)
    total += 0.5M;

if (options.ModalModel != null)
    total += options.ModalModel.HideSelectors?.Count * 0.1M ?? 0;

// Safety Limits
total = Math.Max(total, 1);
total = Math.Min(total, 20);

return (int)Math.Ceiling(total);
}

private static decimal GetAdvancedCost(AdvancedConfigurationModel? adv)
{
    if (adv == null)
        return 0;

    var cost = 0.0M;

    // ⌚ Wait for selector (expensive)
    if (!string.IsNullOrEmpty(adv.WaitForSelector))
        cost += 2.0M;

    // 📄 Headers
    if (adv.Headers != null)

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		99

```

        cost += adv.Headers.Count * 0.15M;

// 🍪 Cookies
if (adv.Cookies != null)
    cost += adv.Cookies.Count * 0.2M;

// 🚫 Resource blocking (reduces cost)
if (adv.BlockResources != ResourceBlockOptions.None)
{
    if (adv.BlockResources.HasFlag(ResourceBlockOptions.Images))
        cost -= 0.2M;

    if (adv.BlockResources.HasFlag(ResourceBlockOptions.Fonts))
        cost -= 0.1M;

    if (adv.BlockResources.HasFlag(ResourceBlockOptions.Media))
        cost -= 0.3M;

    if (adv.BlockResources.HasFlag(ResourceBlockOptions.Scripts))
        cost -= 0.4M;

    if (adv.BlockResources.HasFlag(ResourceBlockOptions.Stylesheets))
        cost -= 0.1M;
}

return cost;
}

private static long CalculatePixels(ScreenshotOptionsModel options)

```

		Погойдаш І.С.			ІПЗ.КР.Б-121-26-ІЗ	Арк.
		Єфремов Ю.М.				100
Змн.	Арк.	№ докум.	Підпис	Дата		

```

{
    int width = DefaultWidth, height = DefaultHeight;

    if (options.Clip != null)
    {
        width = options.Clip.Width;
        height = options.Clip.Height ?? InputLimits.ClipModel.MaxHeight;
    }

    return (long)width * height;
}

private static bool IsFullPage(ScreenshotOptionsModel options)
    => options.Clip != null && options.Clip.Height == null;

private static decimal GetLoadingMultiplier(ContentLoadingOptions
options)
{
    return options switch
    {
        ContentLoadingOptions.None => 1.0M,
        ContentLoadingOptions.ScrollToTheEndOfThePage => 1.3M,
        ContentLoadingOptions.WaitForRequestsToComplete => 1.5M,
        ContentLoadingOptions.All => 1.8M,
        _ => throw new NotImplementedException()
    };
}
}

```

```

public class SubscriptionManager(
    ISubscriptionRepository screenshotRepository,
    IUserContextAccessor userContextAccessor,
    ICacheManager cacheManager,
    IUserCacheService userCache) : ISubscriptionManager
{
    private readonly ISubscriptionRepository _screenshotRepository =
screenshotRepository;

    private readonly AsyncReadWriteLockManager<Guid> _lockPerUser = new();

    private readonly ICacheManager _cacheManager = cacheManager;

    private readonly IUserCacheService _userCache = userCache;

    private Task<IDisposable> EnterLockAsync(Guid userId, bool isRead = true)
    {
        return isRead
            ? _lockPerUser.EnterReadAsync(userId)
            : _lockPerUser.EnterWriteAsync(userId);
    }

    public async Task<ConditionalResult>
CanMakeScreenshotAsync(CanMakeScreenshotModel model, Guid? userId = null)
    {
        userId ??= userContextAccessor.GetCurrentUser().UserInfo.Id;

        using var _ = await EnterLockAsync(userId.Value);

        return await _screenshotRepository.CanMakeScreenshotAsync(model,
userId.Value);
    }
}

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				102
Змн.	Арк.	№ докум.	Підпис	Дата		

```

    }

    public async Task<Result> ScreenshotWasMadeAsync(MakeScreenshotModel
model, Guid? userId = null)
    {
        userId ??= userContextAccessor.GetCurrentUser().UserInfo.Id;

        using var _ = await EnterLockAsync(userId.Value, isRead: false);

        var result = await
_screenshotRepository.CanMakeScreenshotAsync(new(model.PointsCost),
userId.Value);

        if (!result.IsSuccess)
            return Result.Error(result.ErrorMessage!);

        if (!result.Value!)
            return Result.Error("You cannot make screenshot any more because
you ran out of available screenshots");

        var subscription = await
_screenshotRepository.ScreenshotWasMadeAsync(model, userId.Value);

        if (!subscription.IsSuccess)
            return Result.Error(subscription.ErrorMessage!);

        await InvalidateUserCacheAsync(userId.Value);

        return Result.Success;
    }

```

```

        public async Task<Result> RedeemScreenshotAsync(RedeemScreenshotModel
model, Guid? userId = null)
    {
        userId ??= userContextAccessor.GetCurrentUser().UserInfo.Id;

        using var _ = await EnterLockAsync(userId.Value, isRead: false);

        return await _screenshotRepository.RedeemScreenshotAsync(model,
userId.Value);
    }

    private async Task InvalidateUserCacheAsync(Guid userId)
    {
        await _cacheManager.RemoveAsync(_userCache.Profile(userId));
    }
}

public class AuthorizationManager(
    IOptions<AuthorizationConfiguration> options,
    ILogger<AuthorizationManager> logger,
    ITokenReuseDetectionService tokenReuseDetectionService) :
IAuthorizationManager
{
    private readonly AuthorizationConfiguration _config = options.Value;

    private readonly ILogger<AuthorizationManager> _logger = logger;

    private readonly ITokenReuseDetectionService _tokenReuseDetectionService
= tokenReuseDetectionService;

    public async Task<ConfirmationData?> ValidateConfirmationToken(string
token)
    {
        try

```

```

{
    var key = Encoding.UTF8.GetBytes(_config.Secret);
    var validationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidAudience = _config.Audience,
        ValidIssuer = _config.Issuer,
        IssuerSigningKey = new SymmetricSecurityKey(key)
    };

    var tokenHandler = new JwtSecurityTokenHandler();
    var userClaims = tokenHandler.ValidateToken(token,
validationParameters, out _);

    if (userClaims.GetTokenType() is not
Constants.Claims.TokenTypes.Confirmation)

        return null;

    var tokenId =
userClaims.GetClaimValue(Constants.Claims.ConfirmationTokenId);

    if (string.IsNullOrEmpty(tokenId)

        || await
_tokenReuseDetectionService.DetectConfirmationTokenReuseAsync(tokenId))

        return null;

    if
(!int.TryParse(userClaims.GetClaimValue(Constants.Claims.ScreenshotCost), out
int costPoints))

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				105
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        return null;

        var (userId, screenshotId) = (userClaims.GetUserId(),
userClaims.GetClaimValue(Constants.Claims.ScreenshotId));

        if (!userId.HasValue || string.IsNullOrEmpty(screenshotId))
            return null;

        return new(
            userId.Value,
            tokenId,
            screenshotId,
            costPoints
        );
    }
    catch
    {
        return null;
    }
}

public string? GenerateConfirmationToken(ConfirmationData
confirmationData)
{
    try
    {
        var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config.Secret));

        var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);

```

```

        var token = new JwtSecurityToken(
            issuer: _config.Issuer,
            audience: _config.Audience,
            claims: confirmationData.GetConfirmationTokenClaims(),
            expires:
DateTime.UtcNow.AddMinutes(_config.ExpiredInMinutes[AuthorizationType.Confirm
ationToken]),
            signingCredentials: credentials
        );

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to generate confirmation token.");
        return null;
    }
}
}

```

```

public class ScreenshotManager(
    IScreenshotRepository screenshotRepository,
    IUserContextAccessor userContextAccessor,
    IScreenshotEntityMapper screenshotEntityMapper,
    ICacheManager cache,
    IScreenshotCacheService screenshotCache)
    : IScreenshotManager
{
    private readonly IScreenshotRepository _screenshotRepository =
screenshotRepository;

```

```

        private readonly IUserContextAccessor _userContextAccessor =
userContextAccessor;

        private readonly IScreenshotEntityMapper _screenshotEntityMapper =
screenshotEntityMapper;

        private readonly ICacheManager _cache = cache;

        private readonly IScreenshotCacheService _screenshotCache =
screenshotCache;

        public async ValueTask<Result<Screenshot>> GetScreenshot(string
screenshotId)
        {
            var result = await GetScreenshotInternal(screenshotId);

            return FormatScreenshotResult(result);
        }

        private async ValueTask<Result<ScreenshotEntity>>
GetScreenshotInternal(string screenshotId)
        {
            var key = _screenshotCache.ById(screenshotId);

            var result = await _cache.GetOrSetAsync(
                key,
                () => _screenshotRepository.GetScreenshot(screenshotId),
                CacheOptions.Screenshot.Entry);

            return result;
        }

        public async ValueTask<Result<PaginationResult<Screenshot>>>
GetScreenshots(
            ScreenshotPaging? paging = null,
            Guid? userId = null)

```

```

{
    userId ??= _userContextAccessor.GetCurrentUser().UserInfo.Id;

    if (!ShouldCacheList(paging))
        return await GetScreenshotsInternal(paging, userId.Value);

    var cachePaging = paging is not null
        ? new PagingCacheModel(paging.Page, paging.PageSize)
        : null;

    var key = _screenshotCache.List(userId.Value, cachePaging);

    return await _cache.GetOrSetAsync(
        key,
        () => GetScreenshotsInternal(paging, userId.Value),
        CacheOptions.Screenshot.List);
}

public async Task<Result> DeleteAsync(IReadOnlyCollection<string> ids)
{
    if (ids.Count == 0)
        return Result.Success;

    const int maxAttempts = 3;

    foreach (var id in ids.Take(maxAttempts))
    {
        var result = await GetScreenshotInternal(id);
    }
}

```

```

        if (!result.IsSuccess)
            continue;

        var userId = result.Value!.UserId;

        await _screenshotRepository.DeleteAsync(ids);
        await InvalidateListCacheAsync(userId);

        return Result.Success;
    }

    return Result.Error($"deletion process was aborted because more than
{maxAttempts} products weren't found in the database");
}

    public async Task<Result<Screenshot>> MakeAsync(ScreenshotCreateModel
screenshot)
    {
        var entity = await _screenshotRepository.MakeAsync(screenshot);

        var result = FormatScreenshotResult(entity);

        if (result.IsSuccess)
            await InvalidateListCacheAsync(entity.Value!.UserId);

        return result;
    }

    public async Task<Result<Screenshot>> UpdateAsync(ScreenshotUpdateModel
screenshot)

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				110
Змн.	Арк.	№ докум.	Підпис	Дата		

```

{
    var entity = await _screenshotRepository.UpdateAsync(screenshot);

    var result = FormatScreenshotResult(entity);

    if (result.IsSuccess)
        await InvalidateListCacheAsync(entity.Value!.UserId);

    return result;
}

public async Task<Result<Screenshot>> UpdateStateAsync(
    string screenshotId,
    ScreenshotState state)
{
    var entity = await _screenshotRepository.UpdateStateAsync(
        screenshotId,
        state);

    var result = FormatScreenshotResult(entity);

    if (result.IsSuccess)
        await InvalidateListCacheAsync(entity.Value!.UserId);

    return result;
}

private async Task<Result<PaginationResult<Screenshot>>>
GetScreenshotsInternal(

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				ІІІ
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        ScreenshotPaging? paging,
        Guid userId)
    {
        var screenshots = await _screenshotRepository.GetScreenshots(
            paging,
            userId);

        if (!screenshots.IsSuccess)
            return Result<PaginationResult<Screenshot>>
                .Error(screenshots.ErrorMessage!);

        var data = screenshots.Value!;

        var result = new PaginationResult<Screenshot>(
            data.TotalCount,
            [.. data.Items.Select(_screenshotEntityMapper.FromEntity)]);

        return Result<PaginationResult<Screenshot>>
            .Success(result);
    }

    private async Task InvalidateListCacheAsync(Guid userId)
        => await _screenshotCache.InvalidateUserAsync(userId);

    private static bool ShouldCacheList(ScreenshotPaging? paging)
    {
        if (paging is null)
            return true;
    }

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				112
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        return paging.Page == 1

        && paging.PageSize <= 50

        && string.IsNullOrEmpty(paging.Query)

        && (paging.CategoryIds is null || paging.CategoryIds.Count == 0);
    }

    private Result<Screenshot> FormatScreenshotResult(
        Result<ScreenshotEntity> entity)
    {
        if (!entity.IsSuccess)
            return Result<Screenshot>.Error(entity.ErrorMessage!);

        return Result<Screenshot>.Success(
            _screenshotEntityMapper.FromEntity(entity.Value!));
    }
}

public class MessageBrokerManager(ILogger<MessageBrokerManager> logger,
    IMessageBrokerChannelManager messageBrokerChannelManager) :
    IMessageBrokerManager
{
    private readonly ILogger<MessageBrokerManager> _logger = logger;

    private readonly IMessageBrokerChannelManager
    _messageBrokerChannelManager = messageBrokerChannelManager;

    public async Task<bool> SendMessageAsync<T>(T message, string routeKey,
        CancellationToken cancellationToken = default) where T : class
    {
        try
        {

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				
Змн.	Арк.	№ докум.	Підпис	Дата		113

```

        using var timeoutCts = new
CancellationTokensource(TimeSpan.FromSeconds(120));

        using var combinedCancellationToken = cancellationToken ==
default
        ? timeoutCts
        :
CancellationTokensource.CreateLinkedTokenSource(timeoutCts.Token,
cancellationToken);

        var channel = await
_messageBrokerChannelManager.GetChannelAsync(combinedCancellationToken.Token)
;

        await channel.PublishAsync(message, routeKey,
combinedCancellationToken.Token);
    }

    catch (Exception ex)
    {
        _logger.LogError(ex, "Error sending message");
        return false;
    }

    return true;
}

}

public class MassTransitChannelManager(IPublishEndpoint publishEndpoint) :
IMessageBrokerChannelManager
{
    private readonly IPublishEndpoint _publishEndpoint = publishEndpoint;

```

```

    public Task<IBrokerChannel> GetChannelAsync(CancellationToken
cancellationTokен = default)

    {

        IBrokerChannel channel = new
MassTransitBrokerChannel(_publishEndpoint);

        return Task.FromResult(channel);

    }

}

{

    private readonly IPublishEndpoint _publishEndpoint = publishEndpoint;

    public async Task PublishAsync<T>(

        T message,

        string routeKey,

        CancellationToken cancellationTokен = default) where T : class

    {

        await _publishEndpoint.Publish(message, context =>

        {

            context.Headers.Set("routeKey", routeKey);

        }, cancellationTokен);

    }

}

```

		Погойдаш І.С.			ІІЗ.КР.Б-121-26-ІІЗ	Арк.
		Єфремов Ю.М.				115
Змн.	Арк.	№ докум.	Підпис	Дата		