**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА**

**Факультет математики та інформатики**
**кафедра математичного моделювання**

# Пуш-система для мобільних додатків

**Кваліфікаційна робота**

**Рівень вищої освіти – другий (магістерський)**

***Виконав:***
студент 6 курсу, 601 групи
**Добжинецький Максим Романович**
***Керівник:***
доцент, канд. фіз.-мат. наук
Готинчан Т.І.

***До захисту допущено***
***на засіданні кафедри***
***протокол № 5 від 26 листопада 2024 р.***
***Зав. кафедрою _____ проф. Черевко І.М.***

**Чернівці – 2024**

### *Анотація*

Розроблено вебсервіс, що дозволяє користувачам налаштовувати пуш-нотифікації для мобільних додатків на платформах Android та iOS.

Цей вебсервіс значно спрощує процес налаштування пуш-нотифікацій та надає користувачам зручний інтерфейс для керування аудиторіями та повідомленнями. Було використано такі технології, як: мова програмування Python, фреймворк Django, база даних Postgresql, Docker, мова розмітки HTML, мова стилів CSS, скриптова мова програмування JavaScript, асинхронна черга Celery та розподілене сховище Redis.



***Ключові слова:*** Вебсервіс, пуш-система, HTML, CSS, JavaScript, Django, Python, Celery, Redis, Docker, Postgresql.



### *Abstract*

A website has been developed that allows users to configure push notifications for mobile applications on Android and iOS platforms.

This website greatly simplifies the process of setting up push notifications and provides users with a convenient interface for managing audiences and messages. The following technologies were used: Python programming language, Django framework, Postgresql database, Docker, HTML markup language, CSS style language, JavaScript programming language, Celery asynchronous queue and Redis distributed storage.



**Keywords:** Website, push system, HTML, CSS, JavaScript, Django, Python, Celery, Redis, Docker, Postgresql.


Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів наукових досліджень інших авторів мають посилання на відповідне джерело.

_____ М. Р. Добжинецький

# Зміст

Вступ ..... 4
1 Проблематика тематики та постановка завдання ..... 6
1.1 Проблематика тематики ..... 6
1.2 Аналіз подібних вебсервісів та підготовка моделі програмних продуктів ..... 7
1.3 Постановка завдання ..... 8
2 Огляд технологій розробки ..... 10
2.1 Мова програмування Python ..... 10
2.1 Фреймворк Django ..... 11
2.2 Celery ..... 13
2.4 Redis ..... 15
2.5 Postgresql ..... 16
2.6 Використання HTML, CSS та JavaScript при розробці вебсервісів ..... 18
2.7 Docker ..... 20
3 Програмна реалізація ..... 22
3.1 Розробка схема розсилки пуш-нотифікацій користувачам ..... 22
3.2 Розробка функціоналу для створення офферів ..... 24
3.3 Розробка функціоналу для створення аудиторій ..... 27
3.4 Розробка функціоналу для створення сегментації ..... 31
3.5 Розробка функціоналу для створення пуш-нотифікацій ..... 32
3.6 Розробка функціоналу для створення метрик ..... 36
3.7 Розробка функціоналу для надсилання тестової пуш-нотифікації ..... 39
4 Інструкція користувачу ..... 43
Висновки ..... 55
Перелік використаних джерел ..... 56
Додатки ..... 58
Додаток А ..... 58
Додаток Б ..... 61

## Вступ

**Актуальність роботи.** У сучасному цифровому світі ефективна комунікація з користувачами є ключовим елементом успіху для мобільних додатків. Зростання конкуренції та зміна вимог клієнтів вимагають від розробників додатків постійного вдосконалення засобів взаємодії з аудиторією. Особливо актуальною є автоматизація та налаштування пуш-нотифікацій, які дозволяють швидко і зручно доставляти важливу інформацію користувачам. У зв'язку з цим, було розроблено вебсервіс, що дозволяє налаштовувати пуш-нотифікації для мобільних додатків на платформах Android та iOS. Цей інструмент надає можливість швидко та легко керувати аудиторіями, створювати та тестувати пуш-кампанії, а також отримувати детальну статистику про ефективність повідомлень.

**Мета.** Метою кваліфікаційної роботи є розробка вебсервісу, що забезпечує користувачам можливість налаштовувати та керувати пуш-нотифікаціями для мобільних додатків на платформах Android та iOS. Основні завдання, що стоять перед цією розробкою, включають:

- ✓ створення зручного інтерфейсу для керування аудиторіями та повідомленнями;
- ✓ реалізація можливості тестування кількох варіантів пуш-нотифікацій;
- ✓ забезпечення доступу до статистики та аналітики по всіх відправлених повідомленнях;
- ✓ інтеграція з мобільними платформами для забезпечення безперебійної роботи системи.

**Завдання роботи.** З мети випливає, що завданням кваліфікаційної роботи є:

- ✓ розробити інтерфейс для керування аудиторіями користувачів;
- ✓ створити можливость додавання та редагування пуш-нотифікацій;
- ✓ реалізовати функціонал тестування різних варіантів повідомлень;
- ✓ забезпечити доступ до статистики за всіма відправленими пуш-нотифікаціями;

✓ інтегровати вебсервіс з мобільними додатками на платформах Android та iOS.

При розробці вебсервісу були використані Python та Django для створення backend частини. Для відправлення пуш-сповіщень було використано Celery. Під час роботи над вебсервісом використовувались мова гіпертекстової розмітки HTML, каскадні таблиці стилів CSS, скриптова мова програмування JavaScript.

**Об’єкт дослідження** – Об'єктом дослідження є сучасні підходи та технології у розробці вебсервісу для налаштування пуш-нотифікацій, а також технології інтеграції з мобільними додатками на платформах Android та iOS.

**Практичне застосування** полягає у можливості використання розробленого вебсервісу для автоматизації налаштування та керування пуш-нотифікаціями, що покращить взаємодію з користувачами мобільних додатків, підвищить ефективність комунікаційних кампаній та оптимізує керування аудиторіями.

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел та додатків.

У першому розділі описно тематику кваліфікаційної роботи, а також уже деякі створені подібні вебсервіси. Другий розділ містить у собі огляд обраних для розробки технологій та їхні можливостей. Третій розділ складається з опису етапів розробки вебсервісу. Четвертий розділ слугує інструкцією для користувача з розробленої програмної реалізації.

Під час написання дипломної роботи використано інтернет ресурси за посиланнями [1-12].

# 1 Проблематика тематики та постановка завдання

## 1.1 Проблематика тематики

Автоматизація роботи з пуш-нотифікаціями має вирішальне значення для успішного функціонування мобільних додатків. Ось декілька основних причин, чому важливо автоматизувати керування пуш-нотифікаціями:

1) підвищення ефективності: Автоматизація дозволяє оптимізувати рутинні процеси та швидше надсилати повідомлення користувачам. Це значно підвищує продуктивність та дозволяє спрямовувати ресурси на стратегічні аспекти розробки додатків;
2) точність та мінімізація помилок: Автоматизовані системи рідше допускають помилки, що можуть виникнути через людський фактор. Це особливо важливо в управлінні пуш-нотифікаціями, де точність є критичною для забезпечення належного користувацького досвіду;
3) швидка доставка повідомлень: Автоматизовані системи дозволяють швидко доставляти повідомлення, що важливо для забезпечення актуальної інформації користувачам та підтримки їхньої залученості;
4) персоналізований маркетинг: Автоматизовані системи можуть надавати детальну інформацію про користувачів, що дозволяє проводити персоналізовані маркетингові кампанії та створювати цільові повідомлення для конкретних груп користувачів;
5) аналіз та звітність: Автоматизовані системи надають можливість збирати та аналізувати великі обсяги даних про взаємодію користувачів з повідомленнями. Це допомагає приймати обґрунтовані рішення, виявляти тенденції та оптимізувати стратегії комунікації;
6) зниження витрат: Автоматизація процесів дозволяє значно знизити витрати на робочу силу та мінімізувати втрати, пов'язані з помилками та неправильним керуванням повідомленнями;

Загалом, автоматизація керування пуш-нотифікаціями дозволяє підвищити ефективність комунікації з користувачами, покращити обслуговування та залишатися конкурентоспроможним на ринку мобільних додатків.

## 1.2 Аналіз подібних вебсервісів та підготовка моделі програмних продуктів

Переглянувши різні вебсервіси, зокрема, які надають можливість керування пуш-нотифікаціями, можна виділити сильні та слабкі сторони кожного з них. Наприклад, сайт OneSignal [1] вражає зручністю інтерфейсу та можливістю легкої інтеграції з мобільними додатками. Його кольорова гама приємна для очей, усі основні функції розташовані логічно, що сприяє швидкому освоєнню інструменту користувачами.

Інший приклад, Firebase Cloud Messaging [2], також пропонує зручний інтерфейс та великий функціонал для керування пуш-нотифікаціями. Однак, для новачків він може здатися дещо складним через велику кількість налаштувань та можливостей. Кольорова гама та організація елементів інтерфейсу добре продумані, що допомагає зменшити час на навчання.

Вебсервіс Leanplum [3] орієнтований на маркетингові кампанії з можливістю сегментації аудиторії та персоналізації. Реалізовано високу гнучкість у плануванні кампаній і розширені можливості для автоматизації. З недоліків це: складність використання для нефахівців у сфері маркетингу; ціни, які не завжди підходять для малого бізнесу.

На основі аналізу цих та інших вебсервісів було вирішено створити вебсервіс для керування пуш-нотифікаціями, що поєднує найкращі практики.

Основні принципи розробки включають мінімалізм у дизайні, використання пастельних тонів для кольорової гами, чіткі та читабельні шрифти, а також зручну організацію елементів інтерфейсу.

### 1.3 Постановка завдання

Завданням проєкту є створення комплексної системи керування пуш-нотифікаціями для мобільних додатків, зокрема:

1) розробка і впровадження механізму для створення та керування офферами для різних операційних систем;
2) створення інтерфейсу для пуш-нотифікацій, що дозволить цільовій аудиторії ефективно отримувати інформацію;
3) розробка засобів для моніторингу і аналізу ефективності відправлених пуш-повідомлень. Користувацький інтерфейс для кожного з цих компонентів має бути зрозумілим та простим у використанні для забезпечення зручності кінцевого користувача.

Детальніше завдання можна розділити на наступні підзавдання:

1) створення інтерфейсу для роботи з офферами в системах iOS та Android, де користувачі зможуть легко додавати, редагувати та видаляти оффери. Важливою частиною є система номенклатури офферів, яка має бути уніфікована для зручності керування;
2) розробка скедюлера для планування відправки пуш-нотифікацій, який буде інтегрований з основною системою. Цей інструмент дозволить користувачам заздалегідь налаштовувати графіки розсилки, адаптовані до специфіки їхньої аудиторії та маркетингових кампаній. Скедюлер надасть можливість встановлювати частоту, час і дату відправки, що сприятиме більш ефективному досягненню цільової аудиторії у найбільш відповідний час.
3) реалізація вебсервісу для аналітики та моніторингу відправлених пуш-повідомлень. Цей портал включатиме інструменти для аналізу даних, графічне представлення статистики, історію відправлених повідомлень, а також керування кампаніями.

Цей підхід забезпечить не тільки ефективність у роботі з кінцевими користувачами, але й дасть можливість здійснювати глибокий аналіз

ефективності проведених кампаній, що є важливим для оптимізації маркетингових витрат.

Отже, основний функціонал вебсервісу включатиме:

- Дашборд: Відображення статистики по застосунках у вигляді діаграм.
- Аудиторія: Створення та редагування аудиторій користувачів.
- Пуші: Додавання нових пуш-повідомлень для аудиторій.
- Оффери: Додавання назв офферів.
- Пристрої: Перегляд активних та неактивних пристроїв користувачів.
- Метрики: Статистика по всіх відправлених пуш-повідомленнях.
- IOS: Аналогічний функціонал, як і в розділі Android, але для iOS.
- Метрики загальних пушів: Інформація про загальні пуші по вашому пуш сабу.
- Тести: Можливість тестування кількох варіантів пушів.

# 2 Огляд технологій розробки

## 2.1 Мова програмування Python

Мова програмування Python – це високорівнева мова програмування загального призначення, яка є однією з найпопулярніших у світі завдяки своїй простоті, читабельності та широким можливостям [4].

Основні характеристики Python:

- Простота та читабельність: Python має простий і зрозумілий синтаксис, що робить його легким для вивчення та використання. Завдяки цьому програмісти можуть швидше писати код і легше його розуміти.
- Високий рівень абстракції: Python дозволяє програмістам зосередитися на вирішенні проблеми, а не на деталях реалізації, що підвищує продуктивність розробки.
- Інтерпретована мова: Python виконується інтерпретатором, що дозволяє швидко тестувати та відлагоджувати код. Це також полегшує роботу з різними платформами та середовищами.
- Велика стандартна бібліотека: Python має багату стандартну бібліотеку, яка надає численні модулі та функції для виконання різноманітних задач, від обробки тексту до роботи з мережею.
- Розширюваність: Python легко інтегрується з іншими мовами програмування, такими як C або Java. Це дозволяє використовувати Python у складних проєктах та системах.
- Кросплатформність: Python працює на багатьох платформах, включаючи Windows, macOS, Linux та інші операційні системи, що робить його універсальним інструментом для розробки.

Функціональні можливості Python:

- Гнучкість розробки: Python підтримує кілька парадигм програмування, включаючи об'єктно-орієнтоване, функціональне та імперативне програмування.

- Велика екосистема: Існує величезна кількість сторонніх бібліотек і фреймворків для Python, які охоплюють різні області, такі як веброзробка (Django, Flask), наука про дані (NumPy, pandas, SciPy), машинне навчання (TensorFlow, PyTorch) та інші.
- Автоматизація: Python часто використовується для автоматизації рутинних задач, написання скриптів та створення утиліт для системного адміністрування.
- Розробка вебдодатків: Завдяки фреймворкам як Django та Flask, Python широко використовується для створення динамічних вебсервісів і вебдодатків.
- Наука про дані та машинне навчання: Python є одним з найбільш популярних мов у спільноті науковців та інженерів, які працюють з даними та машинним навчанням, завдяки таким бібліотекам як NumPy, pandas, scikit-learn та багатьом іншим.

Використання Python:

- Веброзробка: Використання фреймворків Django та Flask для створення вебдодатків та API.
- Аналіз даних: Використання бібліотек для обробки, аналізу та візуалізації даних.
- Автоматизація: Написання скриптів для автоматизації системних задач, таких як обробка файлів, керування серверами тощо.
- Наукові дослідження: Використання у дослідженнях для моделювання, симуляцій та обробки даних.

## 2.1 Фреймворк Django

Django – це високорівневий вебфреймворк, який дозволяє швидко та ефективно розробляти вебдодатки мовою програмування Python. Django був розроблений з метою спрощення створення складних, багатофункціональних

вебдодатків, зосереджуючи увагу на автоматизації рутинних задач та забезпеченні високого рівня безпеки [5].

Основні характеристики Django:

- Принцип "Батарейки включені": Django надає безліч убудованих функціональностей, які покривають більшість потреб веброзробки – від автентифікації користувачів до керування адміністративним інтерфейсом.
- Швидкість розробки: Завдяки своєму структурованому підходу та багатим вбудованим інструментам, Django дозволяє швидко переходити від початкової ідеї до готового продукту.
- Безпека: Django забезпечує високий рівень безпеки, включаючи захист від найпоширеніших вразливостей, таких як SQL-ін'єкції, міжсайтовий скриптинг (XSS) та підробка міжсайтових запитів (CSRF).
- Масштабованість: Django може використовуватися для розробки як невеликих сайтів, так і великих вебдодатків з високим трафіком.

Функціональні можливості Django:

- ORM (Object-Relational Mapping): Django використовує потужний ORM для роботи з базами даних, що дозволяє розробникам маніпулювати даними за допомогою Python-коду, а не SQL-запитів.
- Адміністративний інтерфейс: Django автоматично створює адміністративний інтерфейс для керування моделями даних, що значно спрощує процес адміністрування додатка.
- Маршрутизація URL: Django надає зручні засоби для визначення маршрутів URL, що дозволяє легко налаштовувати структуру додатку.
- Форми та валідація: Django має вбудовані інструменти для створення та валідації форм, що значно полегшує роботу з введенням даних користувачем.

- Шаблони: Django використовує потужну систему шаблонів, яка дозволяє легко створювати динамічний HTML-код з використанням простого синтаксису.
- Кешування: Django має вбудовані засоби для кешування, що дозволяє значно підвищити продуктивність вебдодатків.

Використання Django:

- Вебсервіси: Django підходить для створення різноманітних вебсервісів, від блогів та інформаційних порталів до корпоративних сайтів та соціальних мереж.
- Вебдодатки: Завдяки своїй масштабованості та багатим можливостям, Django використовується для розробки складних вебдодатків, які обробляють великий обсяг даних та високий трафік.
- API: З використанням Django REST Framework, Django є потужним інструментом для створення RESTful API, що дозволяє легко інтегруватися з іншими системами та сервісами.

## 2.2 Celery

Celery – це розподілена система черг для виконання асинхронних завдань, яка використовується для керування і виконанням фонових завдань у додатках. Вона написана мовою програмування Python і є високоефективним та гнучким інструментом для організації асинхронних процесів [6].

Основні характеристики Celery:

- Асинхронність: Celery дозволяє виконувати завдання у фоновому режимі, що підвищує продуктивність і швидкість основного додатка, розвантажуючи його від тривалих операцій.
- Розподіленість: Celery підтримує розподілену архітектуру, що дозволяє розподіляти завдання між декількома робітниками (workers) та масштабувати систему для обробки великого обсягу завдань.

- Надійність: Celery забезпечує надійне виконання завдань завдяки підтримці зберігання стану завдань, повторного виконання у випадку помилок та керування пріоритетами завдань.
- Простота інтеграції: Celery легко інтегрується з різними брокерами повідомлень, такими як RabbitMQ, Redis, та іншими, що забезпечує гнучкість у виборі інфраструктури.

Функціональні можливості Celery:

- Планування завдань: Celery дозволяє не лише виконувати завдання асинхронно, але й планувати їх виконання у майбутньому або періодично за розкладом.
- Моніторинг і керування: Celery надає інструменти для моніторингу та керування завданнями в реальному часі, що дозволяє відслідковувати статус виконання завдань та реагувати на проблеми.
- Композиція завдань: Celery дозволяє створювати складні робочі процеси, об'єднуючи завдання у ланцюжки (chains) або групи (groups), які можуть виконуватися послідовно або паралельно.

Використання Celery:

- Фонові завдання: Celery часто використовується для виконання тривалих фонових завдань, таких як обробка зображень, відправка електронних листів, генерування звітів та інших ресурсомістких операцій.
- Розподілені обчислення: Завдяки підтримці розподіленої архітектури, Celery підходить для виконання складних обчислювальних задач, які потребують розподілу між кількома робітниками.
- Планування завдань: Celery використовується для виконання завдань за розкладом, наприклад, для регулярних резервних копій, оновлення даних або періодичних звітів.

### 2.4 Redis

Redis (Remote Dictionary Server) – це швидка, відкрита і в пам’яті система баз даних типу key-value, яка також підтримує структури даних, такі як списки, множини та хеш-таблиці. Завдяки своїм особливостям, Redis широко використовується для кешування, керування чергами повідомлень та інших завдань, що вимагають високої продуктивності [7].

Основні характеристики Redis:

- Висока продуктивність: Redis працює повністю у пам'яті, що забезпечує надзвичайно швидкий доступ до даних і високу пропускну здатність.
- Розширені структури даних: Окрім звичайних ключ-значення, Redis підтримує різноманітні структури даних, такі як списки, множини, відсортовані множини, хеші, бітові масиви та геопросторові індекси.
- Постійність даних: Redis забезпечує можливість збереження даних на диск за допомогою механізмів snapshotting та journaling, що робить його надійним інструментом для зберігання даних.

Функціональні можливості Redis:

- Кешування: Redis часто використовується для кешування даних, що дозволяє зменшити навантаження на основні бази даних та прискорити доступ до часто використовуваних даних.
- Керування сесіями: Завдяки своїй швидкості, Redis є ідеальним рішенням для керування сесіями користувачів у вебдодатках.
- Черги повідомлень: Redis підтримує операції push/pop, що дозволяє використовувати його для реалізації черг повідомлень та систем обробки подій.
- Аналіз даних у реальному часі: Завдяки швидкій обробці даних, Redis можна використовувати для аналізу даних у реальному часі, таких як аналітика трафіку або відстеження поведінки користувачів.

Використання Redis:

- Кешування даних: Використовується для тимчасового зберігання часто запитуваних даних, що зменшує навантаження на бази даних та прискорює доступ до інформації.
- Системи черг: Завдяки підтримці списків і відсортованих множин, Redis широко використовується для реалізації систем черг, що потребують швидкої обробки завдань.
- Керування сесіями: У вебдодатках Redis часто використовується для зберігання сесійних даних користувачів, що забезпечує швидкий доступ і високу продуктивність.

## 2.5 Postgresql

PostgreSQL – це потужна, відкрита система керування реляційними базами даних (СКБД), яка відома своєю надійністю, функціональністю та масштабованістю. Вона широко використовується у різних галузях для зберігання, обробки та керування даними, завдяки своїм розширеним можливостям і відповідності стандартам SQL [8].

Основні характеристики PostgreSQL:

- Відповідність стандартам SQL: PostgreSQL повністю відповідає стандартам SQL, що забезпечує високу сумісність і можливість використання складних запитів.
- Розширюваність: СКБД дозволяє користувачам додавати нові типи даних, функції, оператори та індекси за допомогою різних мов програмування, включаючи PL/pgSQL, PL/Python, PL/Perl та інші.
- Масштабованість: Підтримує великі обсяги даних та високу кількість одночасних користувачів, що робить її придатною для масштабних проєктів.

- Багатоверсійність (MVCC): Технологія багатоверсійної контрольованої конкуренції забезпечує високий рівень паралелізму і ізоляції транзакцій, що сприяє підвищенню продуктивності.
- Реплікація та відмовостійкість: Підтримує різні методи реплікації даних, включаючи асинхронну та синхронну реплікацію, що підвищує відмовостійкість і доступність системи.

Функціональні можливості PostgreSQL:

- Типи даних: Підтримує широкий спектр типів даних, включаючи числові, символьні, часові, просторові (PostGIS) та інші.
- Індексація: Підтримує різні типи індексів, такі як B-Tree, Hash, GiST, SP-GiST, GIN та BRIN, що забезпечує швидкий доступ до даних.
- Зовнішні таблиці: Можливість підключення зовнішніх таблиць для доступу до даних з інших СКБД або файлів.
- Безпека: Розширені механізми автентифікації та авторизації, підтримка SSL для захищеного з'єднання та можливість керування доступом на рівні рядків (Row-Level Security).
- Підтримка JSON: Зручна робота з JSON та JSONB типами даних, що дозволяє зберігати та обробляти напівструктуровані дані.

Використання PostgreSQL:

- Кешування даних: Використовується для тимчасового зберігання часто запитуваних даних, що зменшує навантаження на бази даних та прискорює доступ до інформації.
- Керування сесіями: У вебдодатках PostgreSQL часто використовується для зберігання сесійних даних користувачів, що забезпечує швидкий доступ і високу продуктивність.
- Аналітика та звітність: Використовується для обробки та аналізу великих обсягів даних, завдяки підтримці складних запитів і функцій агрегації.

## 2.6 Використання HTML, CSS та JavaScript при розробці вебсервісів

HTML (HyperText Markup Language) Living standart – це остання та найсучасніша версія мови розмітки, яка використовується для створення та структурування вмісту в Інтернеті. HTML надає розробникам потужні можливості та покращену функціональність порівняно з попередніми версіями [9].

Основні особливості HTML:

- Семантична розмітка: HTML включає нові теги, які забезпечують більш чітку та логічну структуру вебсторінок, що полегшує їх розуміння пошуковими системами.
- Аудіо та відео: Вбудована підтримка мультимедіа за допомогою тегів.
- Локальне збереження: Специфікація Web Storage дозволяє зберігати дані локально в браузері, що підвищує продуктивність і полегшує роботу з локальними даними.
- Мобільні технології: HTML підтримує адаптивний дизайн, що дозволяє створювати вебдодатки, оптимізовані також і для мобільних пристроїв.

CSS (Cascading Style Sheets) – це мова стилів, яка використовується для опису зовнішнього вигляду вебсторінок. CSS визначає кольори, шрифти, розміри, розташування та інші стилі для HTML-елементів [10].

Основні можливості CSS:

- Стилізація тексту: Визначення кольору, розміру, сімейства шрифтів та інших текстових властивостей.
- Розташування елементів: Розміщення елементів на сторінці за допомогою макетів, таких як flexbox та grid.
- Анімація та переходи: Додавання динамічних ефектів до елементів, таких як анімації та плавні переходи.

- Адаптивний дизайн: Використання медіа-запитів для створення адаптивних макетів, які коректно відображаються на різних пристроях.

JavaScript – це мова програмування, яка використовується для створення інтерактивних вебсторінок. Вона дозволяє реалізовувати функції, які покращують взаємодію користувача з вебсервісом, наприклад, динамічне оновлення контенту, анімації та обробка подій [11].

Основні можливості JavaScript:

- Інтерактивність: Додавання взаємодії з користувачем, такої як реакція на натискання кнопок або введення даних.
- Модифікація DOM: Динамічна зміна структури HTML-документа для оновлення контенту без перезавантаження сторінки.
- Асинхронні-запити: Асинхронне завантаження даних з сервера для оновлення контенту без перезавантаження сторінки.
- Клієнтські та серверні додатки: Використання JavaScript для розробки як клієнтських (браузерних), так і серверних (Node.js) додатків.

Переваги використання HTML, CSS, JavaScript:

- Покращена семантика: Більш зрозумілий та інтуїтивний код.
- Зручність для розробників: Нові функції та інструменти полегшують роботу та підвищують продуктивність.
- Зменшення залежності від плагінів: Більша сумісність із сучасними браузерами та відсутність необхідності встановлення додаткових плагінів.
- Покращені можливості дизайну: Сучасні техніки створення інтерфейсів та графіки.
- Зменшення затримок: Підтримка асинхронного завантаження ресурсів, що прискорює завантаження сторінок.

HTML, CSS та JavaScript є основними технологіями сучасної веброзробки, які забезпечують створення багатофункціональних, адаптивних та інтерактивних вебдодатків. Використання цих технологій дозволяє розробникам створювати якісний контент та покращувати користувацький досвід.

## 2.7 Docker

Docker – це платформа для розробки, доставки та запуску програмного забезпечення за допомогою контейнеризації. Вона дозволяє упаковувати програмне забезпечення та всі його залежності в стандартизовані контейнери, які можна розгортати на будь-якому сервері з Docker і запускати в будь-якому середовищі [12].

Основні характеристики Docker:

- Контейнеризація: Docker використовує контейнери для упакування програмного забезпечення та всіх його залежностей у ізольовані середовища, що забезпечує консистентність та незалежність від середовища виконання.
- Стандартизація: Контейнери Docker базуються на відкритих стандартах, що робить їх переносимими між різними середовищами виконання.
- Швидкість і легкість: Запуск та виконання контейнерів Docker є швидкими і ефективними, оскільки вони використовують ядро операційної системи хоста.
- Масштабованість: Docker забезпечує можливість масштабування за допомогою оркестрації контейнерів, такої як Docker Swarm або Kubernetes.
- Керування ресурсами: Docker надає інструменти для керування ресурсами, такими як CPU, пам'ять та мережа, для кожного контейнера окремо.

Функціональні можливості Docker:

- Ізоляція: Контейнери Docker забезпечують ізольоване середовище виконання, що гарантує, що програмне забезпечення працюватиме однаково незалежно від конфігурації хоста.
- Швидкість розгортання: Запуск та розгортання контейнерів Docker займає декілька секунд, що дозволяє швидко розвертати та масштабувати додатки.
- Модульність: Docker забезпечує можливість розділити програмне забезпечення на набір модулів, які можна упаковувати та розгортати окремо.
- Портативність: Контейнери Docker можна розгортати на будь-якому сервері з Docker, незалежно від операційної системи або хост-системи.

Використання Docker:

- Розробка та тестування: Docker дозволяє розробникам упаковувати та розгортати середовища для розробки та тестування програмного забезпечення, що забезпечує консистентність між різними етапами життєвого циклу додатка.
- Розгортання продукції: Docker використовується для розгортання програмного забезпечення у продукційне середовище, що забезпечує стабільність та надійність роботи додатків.
- Оркестрація контейнерів: Docker Swarm або Kubernetes використовуються для оркестрації та керування багатьма контейнерами Docker у розподіленому середовищі.

# 3 Програмна реалізація

## 3.1 Розробка схема розсилки пуш-нотифікацій користувачам

Схема (рис. 3.1) демонструє процес налаштування та відправлення пуш-нотифікацій через систему One Signal з акцентом на використання аудиторії, метрик, тестів та сегментації. Описуючи схему для кваліфікаційної роботи, можна розглянути кожен елемент та його роль у процесі комунікації з користувачами мобільних додатків.

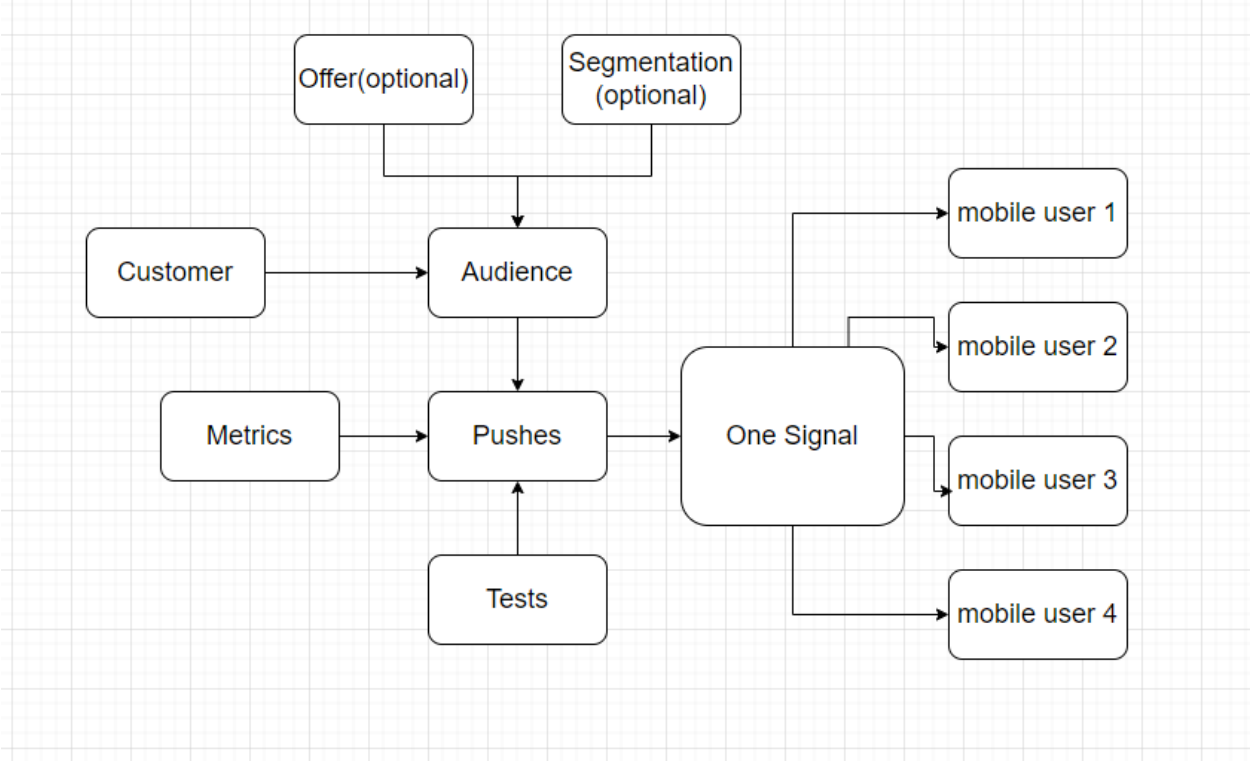



*Рис. 3.1. Схема розсилки пуш-нотифікацій*

Опис схеми:

1. **Customer (Клієнт)**:
   - Це початкова точка взаємодії. Клієнт визначає аудиторію, на яку будуть орієнтовані пуш-нотифікації. Клієнт може надати дані про користувачів або вибрати їх з існуючої бази даних користувачів.
2. **Audience (Аудиторія)**:

- Аудиторія – це група користувачів, на яку будуть націлені повідомлення. Вона формується на основі наданих клієнтом даних та може бути відфільтрована за різними критеріями. У цьому контексті може бути задіяна додаткова сегментація.

3. **Segmentation (Сегментація) (Optional)**:
   - Сегментація є необов’язковим елементом. Вона дозволяє більш точно налаштовувати аудиторію для відправки пуш-нотифікацій. Наприклад, можна вибирати користувачів за певними критеріями, а саме: остання активність, чи був депозит/реєстрація/інсталяція.
4. **Offer (Пропозиція) (Optional)**:
   - Пропозиція також необов'язковий елемент і може використовуватися для формування спеціальних аудиторій з урахуванням конкретних значень «sub», котрі надають конкретні трафік менеджера.
5. **Pushes (Пуш-нотифікація)**:
   - Пуш-нотифікація – це основна функція системи. Вони відправляються аудиторії, сформованій на основі даних клієнта та можливих сегментацій. Push-повідомлення можуть містити інформацію про новини, пропозиції або інші важливі повідомлення для мобільних користувачів.
6. **Tests (Тести)**:
   - Перед відправкою повідомлень проводяться тести для перевірки їх коректності та ефективності. Це важливий етап, щоб переконатися, що повідомлення досягають користувачів у потрібний час і у правильному форматі.
7. **Metrics (Метрики)**:
   - Після відправки повідомлень збираються метрики, що відображають ефективність кампанії. Метрики можуть включати інформацію про кількість відправлених повідомлень, відкриття, взаємодію з повідомленнями та конверсії. На основі цих даних можна аналізувати успішність кампанії та вносити корективи у подальші повідомлення.

8. **One Signal (Система One Signal)**:
    - One Signal виступає як платформа для відправки пуш-нотифікацій мобільним користувачам. Це основний механізм для відправки та моніторингу повідомлень. Вона отримує сформовані пуш-нотифікації та відправляє їх мобільним користувачам.
9. **Mobile users (Мобільні користувачі)**:
    - Це кінцеві одержувачі пуш-нотифікацій. Користувачі отримують повідомлення на свої мобільні пристрої. Важливо враховувати, що повідомлення можуть бути адаптовані залежно від сегментації аудиторії та даних метрик.

**Опис процесу:**

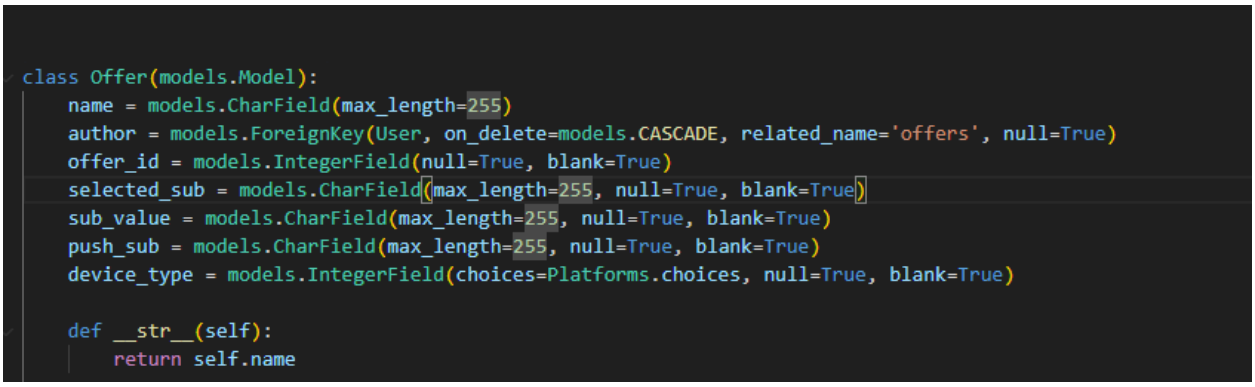
Клієнт визначає аудиторію користувачів, яка може бути сегментована за додатковими критеріями. Додатково до аудиторії можна добавляти оффера, за певним значенням «sub». Після формування аудиторії й сегментації система One Signal відправляє push-повідомлення до мобільних користувачів. Перед відправкою проводяться тести, щоб переконатися, що повідомлення працюють коректно. Після відправки метрики використовуються для аналізу успішності повідомлень, що дозволяє клієнту оцінити ефективність комунікації та зробити необхідні корективи.

**Висновок:**

Ця схема демонструє процес використання системи One Signal для ефективної комунікації з мобільними користувачами через пуш-нотифікації. Важливим аспектом є можливість сегментації аудиторії, проведення тестів перед запуском, а також використання метрик для оцінки результатів і оптимізації кампанії.

### 3.2 Розробка функціоналу для створення офферів

Для початку створюємо модель для офферів (дивитись у додатку А) у базі даних, використовуючи Django ORM.



```python
class Offer(models.Model):
    name = models.CharField(max_length=255)
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='offers', null=True)
    offer_id = models.IntegerField(null=True, blank=True)
    selected_sub = models.CharField(max_length=255, null=True, blank=True)
    sub_value = models.CharField(max_length=255, null=True, blank=True)
    push_sub = models.CharField(max_length=255, null=True, blank=True)
    device_type = models.IntegerField(choices=Platforms.choices, null=True, blank=True)

    def __str__(self):
        return self.name
```

*Рис. 3.2. Представлення моделі оффера*

У системі пуш-нотифікацій важливо не тільки надавати користувачам загальні повідомлення, а й таргетувати аудиторію на основі їхніх особистих даних або поведінкових характеристик. Модель офферів, яку ми використовуємо, дозволяє створювати та фільтрувати користувачів мобільних додатків для більш точного таргетування пуш-повідомлень.

**name**: Це поле відповідає за назву оффера. Кожен оффер може бути асоційований з конкретною акцією або подією. Назва допомагає ідентифікувати оффер у системі для подальшого аналізу або налаштування розсилки.

**author**: Поле представляє автора оффера, який пов'язаний із користувачем системи. Це дозволяє зберігати інформацію про те, хто створив оффер, що особливо важливо в системах із великою кількістю адміністраторів або маркетологів. Таким чином, автор може бути ідентифікований через ForeignKey, що посилається на модель користувача (User).

**offer_id**: Поле, що зберігає ідентифікаційний номер оффера. Воно може бути використане для внутрішнього зберігання та керування офферами. Використання null=True та blank=True вказує на можливість залишати це поле порожнім, якщо оффер не потребує суворої ідентифікації.

**selected_sub**: Це поле є ключовим у фільтрації користувачів. Воно відповідає за вибір певного параметра (наприклад, по останній активності, чи були здійснені депозити/реєстрації/інсталяції), на основі якого буде здійснено фільтрацію аудиторії для надсилання пуш-нотифікацій.

**sub_value**: Значення, яке відповідає вибраному параметру. Це значення надається трафіку менеджером для кожного типу оферів. Це дозволяє більш детально налаштовувати фільтрування аудиторії.

**push_sub**: Це поле містить додаткову інформацію або параметри, які використовуються для сегментування аудиторії. Наприклад, воно може зберігати інформацію про ідентифікаю та зв’язані з ним інші кабінети.

**device_type**: Це поле визначає тип пристрою, на який буде надіслано пуш-нотифікацію. Воно базується на виборі з певних платформ (Platforms.choices). Наприклад, це може бути iOS, Android або Web, що дозволяє таргетувати повідомлення на певні платформи залежно від потреб кампанії.

Надалі було написано сам функціонал для відображення необхідної інформації на шаблонах. Процес відображення сторінки у Django за допомогою шаблонів можна пояснити так: коли користувач уводить запит на певну вебсторінку, наприклад, натискає на посилання або вводить адресу в браузері, сервер обробляє цей запит. Спочатку Django визначає, яка частина коду має реагувати на цей запит, що називається "в'юшкою". В'юшка відповідає за виконання логіки – вона може отримати дані з бази даних або обробити інші дії, необхідні для відповіді на запит. Далі ці дані передаються у шаблон – спеціальний файл, який формує вигляд сторінки. Шаблон містить як статичний HTML-код, так і спеціальні місця, куди підставляються динамічні дані, такі як текст, зображення або інші елементи. Після цього шаблон обробляється сервером, і зібрана сторінка відправляється назад до браузера користувача, де відображається у вигляді готової вебсторінки. Таким чином, Django розділяє логіку програми і процес відображення, роблячи систему зручною та гнучкою для розробки динамічних вебдодатків. Процес відображення сторінки буде аналогічний для всіх інших запитів, зокрема змінюватимуться тільки моделі бази даних, ендпоінти та в’юшки.

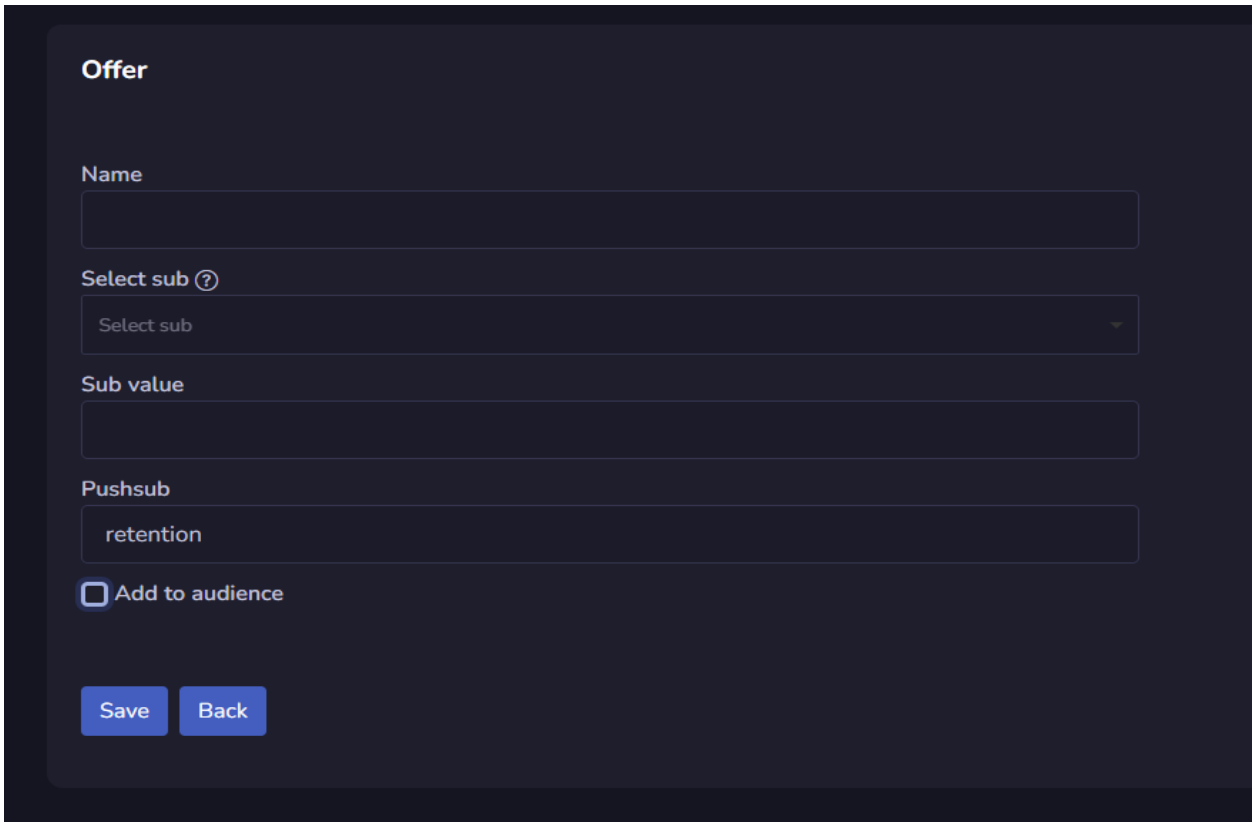



*Рис. 3.3. Сторінка створення оффера*

### 3.3 Розробка функціоналу для створення аудиторій

Після створення оффера, було реалізовано функціонал для взаємодії з аудиторіями. Спочатку було написано клас моделі аудиторії (дивитись у додатку А).

```
class Audience(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(User, on_delete=models.CASCADE, db_index=True)
    device_type = models.IntegerField(choices=Platforms.choices, null=True, blank=True, db_index=True)
    apps_type = models.CharField(max_length=50, choices=AppType.choices, default=AppType.GAMBLING, blank=True, db_index=True)
    geo = models.ManyToManyField(Geo)
    apps = models.ManyToManyField(MobileApp)
    all_applications = models.BooleanField(default=False, blank=True)
    all_geos = models.BooleanField(default=False, blank=True)
    all_push_subs = models.BooleanField(default=False, blank=True)
    push_subs = models.ManyToManyField(Group, blank=True)
    offers = models.ManyToManyField(Offer, blank=True)
    filter_by_offer = models.BooleanField(default=True)
    finished_event = models.CharField(max_length=30, choices=UserEvent.choices, null=True, blank=True)
    unfinished_event = models.CharField(max_length=30, choices=UserEvent.choices, blank=True)
    clicked_push_counter = models.IntegerField(default=0)
    deleted_app = models.IntegerField(default=0)
    received_push_counter = models.IntegerField(default=0)
    users_count = models.PositiveIntegerField(default=0)
    deposit_counter = models.IntegerField(default=0, blank=True)
    is_general = models.BooleanField(default=False, blank=True)

    def __str__(self):
        return f'{self.name}'
```

*Рис. 3.4. Представлення моделі аудиторії*

Модель Audience призначена для формування та управління аудиторіями, що отримують пуш-нотифікації. Кожна аудиторія може бути сегментована за кількома критеріями, такими як тип пристрою, геолокація, активність користувача тощо. Це дозволяє значно підвищити ефективність кампаній, таргетуючи повідомлення на певні групи користувачів, які відповідають конкретним вимогам.

**name**: Це поле представляє назву аудиторії. Назва використовується для зручної ідентифікації аудиторій у системі, що особливо важливо в умовах великої кількості сегментованих груп. Назва має бути короткою та інформативною.

**author**: Поле відповідає за автора аудиторії, тобто користувача, який створив цю групу. Це дозволяє забезпечити прозорість та відповідальність у випадку, якщо потрібно відстежити, хто створив або редагував конкретну аудиторію.

**device_type**: Це поле визначає тип пристрою, на якому користувач отримує пуш-нотифікації. Модель підтримує різні платформи, наприклад, Android, iOS, вебдодатки, що дозволяє більш точно сегментувати аудиторії на основі типу пристроїв, які вони використовують.

**apps_type**: Поле, яке відповідає за тип мобільного додатку. Наприклад, система може розсилати повідомлення тільки користувачам додатків для азартних ігор (Gambling), або іншим категоріям додатків. Це дозволяє налаштувати таргетинг на основі специфічної категорії додатку.

**geo**: Це поле представляє список географічних регіонів, де знаходяться користувачі. Модель підтримує ManyToManyField, що дозволяє одному користувачеві належати до кількох географічних регіонів. Сегментування за географічним принципом є важливою складовою у сучасних рекламних кампаніях.

**apps**: Це поле містить інформацію про мобільні додатки, які використовують користувачі. Сегментування на основі додатків дає можливість

надсилати повідомлення тільки тим користувачам, які користуються певними мобільними програмами.

**all_applications, all_geos, all_push_subs**: Ці поля використовуються для спрощеного сегментування аудиторії. Якщо ці поля встановлено в True, то система не буде фільтрувати користувачів за додатками, геолокацією або підписками на пуш-нотифікації. Це корисно у випадках, коли кампанія орієнтована на максимально широку аудиторію.

**push_subs**: Поле для сегментування аудиторії на основі груп підписників на певні категорії пуш-нотифікацій. Це дозволяє налаштувати більш точний таргетинг на основі вподобань або дій користувачів.

**offers**: Поле ManyToManyField, яке дозволяє прив'язувати кілька офферів до аудиторії. Це корисно, коли одразу кілька кампаній або пропозицій потрібно таргетувати на одну й ту саму групу користувачів.

**filter_by_offer**: Це булеве поле відповідає за фільтрацію аудиторії на основі офферів. Якщо встановлено True, аудиторія буде відфільтрована залежно від наявних офферів, що дозволяє націлити повідомлення на конкретні пропозиції.

**finished_event та unfinished_event**: Ці поля представляють дії користувачів, які завершили або не завершили певні події у додатку. Наприклад, користувач може бути частиною аудиторії, якщо він закінчив реєстрацію (finished_event) або, навпаки, не завершив якийсь процес (unfinished_event). Це дозволяє надсилати нагадування чи спеціальні пропозиції тим, хто не завершив важливу дію.

**clicked_push_counter та received_push_counter**: Поля для відстеження активності користувачів. Вони зберігають кількість отриманих пуш-нотифікацій та кількість натискань на них. Це важливі показники для аналізу ефективності кампаній.

**deleted_app**: Це поле зберігає кількість видалень додатків користувачами з цієї аудиторії. Воно може використовуватися для аналізу причин зниження активності або для попередження користувачів про нові функції додатку.

**users_count**: Поле зберігає кількість користувачів в аудиторії. Це важливий показник для відстеження масштабу кампанії.

**deposit_counter**: Це поле зберігає кількість депозитів, зроблених користувачами в певних додатках. Поле використовується у рекламних кампаніях для залучення активних користувачів до нових пропозицій.

**is_general**: Булеве поле, яке вказує на те, чи є аудиторія загальною або спеціально сегментованою. Загальна аудиторія може бути використана для широких кампаній без суворого таргетингу.

Після створення моделі було реалізовано необхідні ендпоінти і підключено в'юшки до них, на конкретних шаблонах. Ось вигляд сторінки для створення аудиторії.

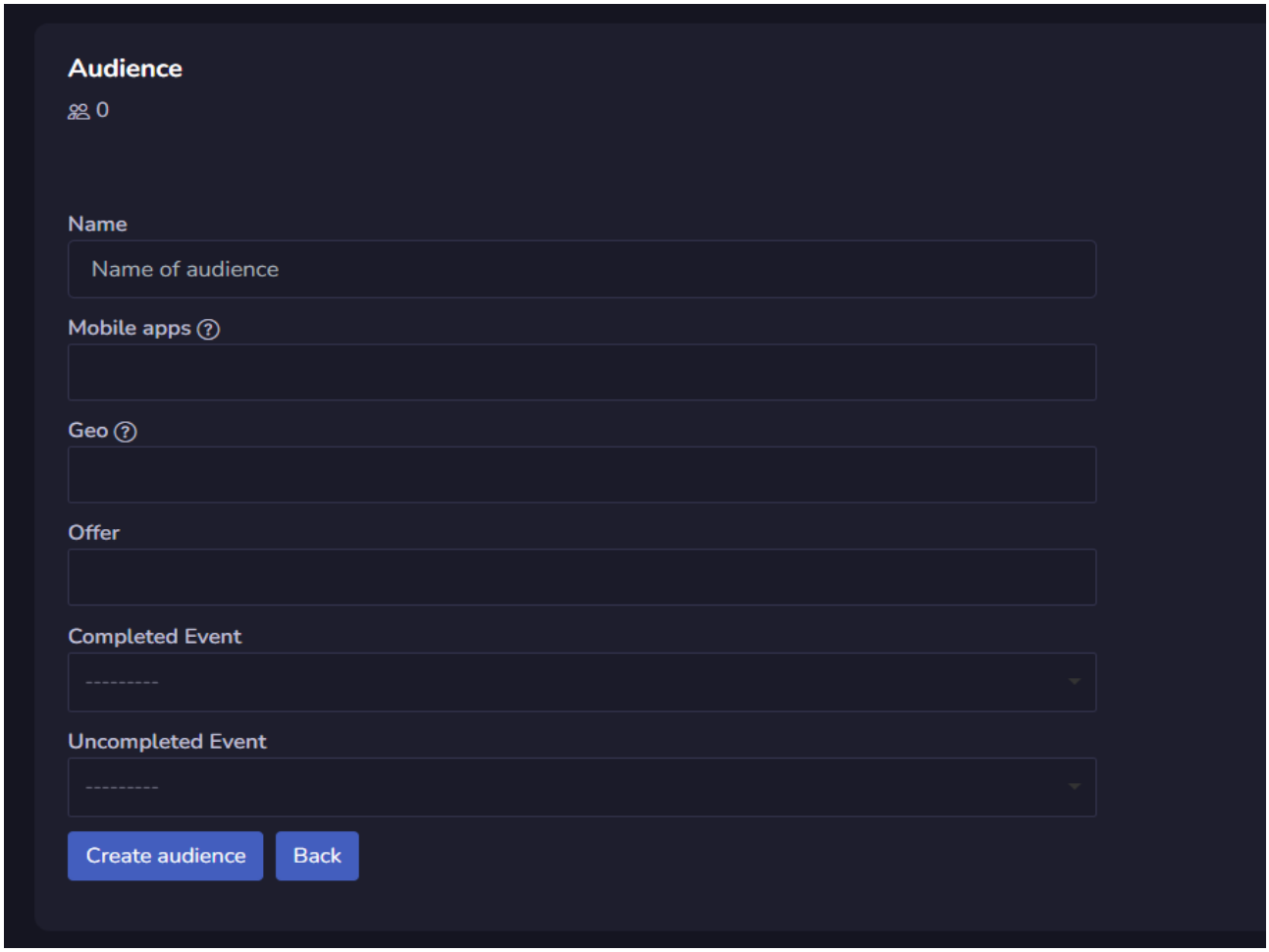



*Рис. 3.5. Сторінка створення аудиторії*

### 3.4 Розробка функціоналу для створення сегментації

За аналогією з попередніми кроками, створено модель AudienceSegmentation. Модель AudienceSegmentation (дивитись у додатку А) використовується для більш точного та гнучкого поділу аудиторії на сегменти на основі певних подій та критеріїв. Вона дозволяє сегментувати користувачів залежно від їхньої поведінки або дій у мобільному додатку, що надає можливість більш ефективно проводити таргетовані маркетингові кампанії.

```
class AudienceSegmentaion(models.Model):
    segment = models.CharField(max_length=150, choices=SegmentType.choices)
    event = models.CharField(max_length=150, choices=EventType.choices, null=True, blank=True)
    comparison = models.CharField(max_length=50, choices=ComparsionType.choices)
    time_period = models.PositiveIntegerField(default=0, null=True, blank=True)

    def __str__(self):
        return (f'{self.segment} - {self.event or ""} {self.comparison} '
                f'{(str(self.time_period) + " days") if self.time_period else ""}')
```

*Рис. 3.6. Реалізація моделі сегментації*

**segment**: Це поле визначає тип сегментації, що використовується для поділу аудиторії. Воно містить список типів сегментів, визначених у SegmentType.choices. Типи сегментації можуть бути різними – від демографічних характеристик до поведінкових даних. Цей параметр дає можливість гнучко налаштовувати сегментацію залежно від потреб кампанії. Наприклад, сегментація може ґрунтуватися на певних діях користувача або на їх поведінкових паттернах.

**event**: Це поле відповідає за подію, яка стала основою для сегментації. Поле використовує список подій, визначених у EventType.choices, і може включати такі події, як реєстрація, покупка, відкриття програми, або будь-які інші важливі взаємодії користувача з додатком. Це дозволяє сегментувати аудиторію залежно від певних дій, що особливо важливо для ретаргетингових кампаній або персоналізованих повідомлень. Якщо це поле залишається порожнім, сегментація може проводитися без врахування конкретної події.

**comparison**: Це поле використовується для порівняння показників або поведінкових характеристик користувачів у сегменті. Поле містить список можливих варіантів порівняння, визначених у ComparsionType.choices, таких як більше, менше, дорівнює тощо. Порівняння дозволяє налаштовувати більш детальні умови для включення користувачів у сегмент. Наприклад, можна вибрати сегмент користувачів, які зробили певну кількість покупок за останній місяць або мають певну активність у додатку.

**time_period**: Це поле визначає часовий період для відстеження подій або активності користувачів. Воно дозволяє обмежити сегментацію певним періодом часу, наприклад, користувачами, які виконали певну дію за останні 7 днів або протягом останнього місяця. Це є важливим аспектом для кампаній, орієнтованих на користувачів, що виявляють активність у певний часовий інтервал.

Після чого реалізовано HTML шаблон з відповідною логікою.

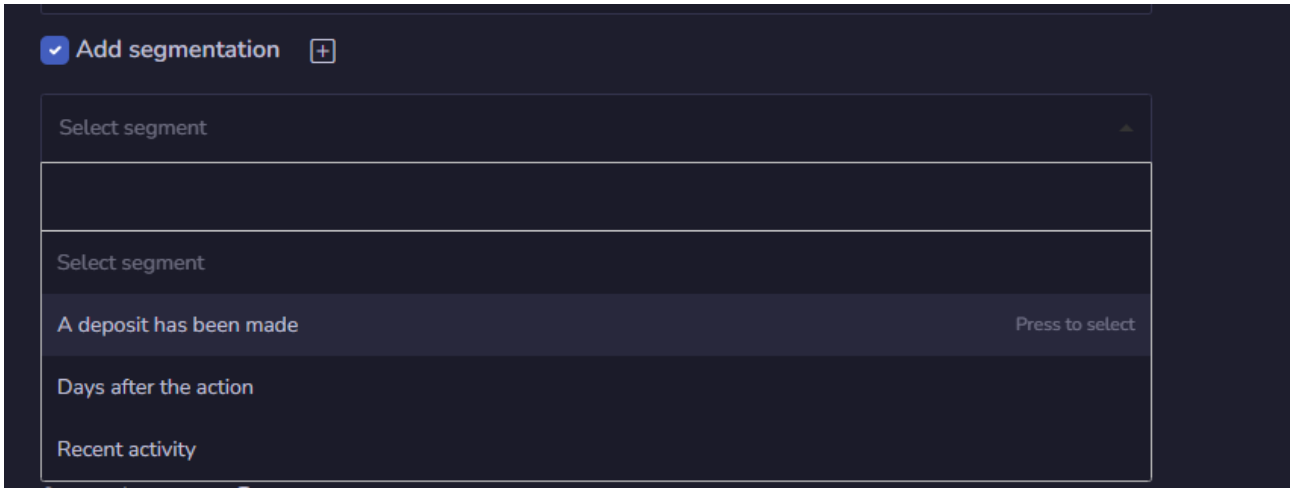



*Рис. 3.7. Сторінка створення сегментації*

### 3.5 Розробка функціоналу для створення пуш-нотифікацій

Нижче представлена модель **Push** (дивитись у додатку Б), яка використовується для створення, налаштування та керування пуш-нотифікаціями.

```python
class Push(models.Model):
    MAILING_DATE = 'date'
    MAILING_DAY = 'day'
    EVENT_PUSH = 'event'

    MAILING_TYPE = (
        (MAILING_DAY, 'Mailing by day'),
        (MAILING_DATE, 'Mailing by date'),
        (EVENT_PUSH, 'Mailing by event')
    )

    title = models.CharField(max_length=150)
    author = models.ForeignKey(User, on_delete=models.CASCADE, db_index=True)
    device_type = models.IntegerField(choices=Platforms.choices, null=True, blank=True)
    audiences = models.ManyToManyField(to=Audience)
    segmentations = models.ManyToManyField(to=AudienceSegmentaion, blank=True)
    mailing_schedule = models.ManyToManyField(to=MailingSchedule, blank=True)
    mailing_date = models.ManyToManyField(to=MailingDate, blank=True)
    event_push = models.ForeignKey(to=EventPush, on_delete=models.SET_NULL, null=True, blank=True)
    header = models.CharField(max_length=150)
    message = models.TextField()
    img_url = models.URLField(max_length=500, null=True, blank=True)
    icon_url = models.URLField(max_length=500, null=True, blank=True)
    endpoint_url = models.URLField(max_length=500, null=True, blank=True)
    push_tasks = models.ManyToManyField(PeriodicTask)
    source_language = models.CharField(max_length=225, null=True)
    is_active = models.BooleanField(default=True)
    is_general = models.BooleanField(default=False)

    def __str__(self):
        return f'{self.header}'
```

*Рис. 3.8. Реалізація моделі пуш-нотифікації*

Модель **Push** призначена для організації та керування пуш-нотифікаціями у мобільних додатках. Вона дозволяє визначати аудиторії, сегментувати їх за різними критеріями, налаштовувати розсилки на основі подій або часу, а також вказувати необхідні повідомлення, зображення і дії для кожної кампанії.

**title**: Поле для назви розсилки. Назва повинна бути інформативною та зрозумілою, оскільки вона використовується для ідентифікації пуш-кампанії всередині системи.

**author**: Це поле визначає автора пуш-розсилки, тобто користувача, який створив цю розсилку. Поле пов'язане з таблицею користувачів за допомогою зовнішнього ключа.

**device_type**: Поле відповідає за тип пристрою, на який буде надсилатися повідомлення (Android, iOS тощо). Це дозволяє сегментувати розсилки на основі використовуваних платформ.

**audiences**: Поле ManyToManyField, яке дозволяє вибирати аудиторії, що отримають пуш-нотифікації. Це дає змогу легко таргетувати певні групи користувачів, базуючись на раніше створених аудиторіях.

**segmentations**: Поле ManyToManyField, що дозволяє додавати сегменти аудиторії на основі поведінки або інших критеріїв. Це дає змогу ще більше деталізувати таргетинг та забезпечити релевантність повідомлень.

**mailing_schedule та mailing_date**: Ці поля відповідають за налаштування часу відправки повідомлень. Пуш-нотифікації можуть бути надіслані в певний день, за певною датою або на основі подій.

**event_push**: Поле, яке дозволяє прив'язати розсилку до певної події (наприклад, завершення покупки, реєстрація, або інша важлива дія користувача). Це дозволяє автоматизувати процес відправки повідомлень у реальному часі залежно від дій користувачів.

**header та message**: Поля для заголовка і тексту повідомлення. **header** — це короткий заголовок, що привертає увагу, а **message** — детальний текст пуш-нотифікації, який містить основну інформацію або заклик до дії.

**img_url та icon_url**: Поля для зображення та іконки, що додаються до повідомлення. Використання зображень та іконок допомагає зробити пуш-нотифікацію більш візуально привабливою та збільшити шанс на залучення користувача.

**endpoint_url**: Поле для URL-адреси, на яку користувач буде перенаправлений після натискання на пуш-нотифікацію. Це може бути вебсторінка, сторінка мобільного додатку або інший ресурс.

**push_tasks**: Поле ManyToManyField, що пов'язане з періодичними завданнями (**PeriodicTask**). Воно використовується для управління повторюваними розсилками або задачами, які автоматично надсилають повідомлення за певним графіком.

**source_language**: Поле для вказівки мови, на якій створена пуш-нотифікація. Це важливо в багатомовних додатках, щоб забезпечити релевантність повідомлень для користувачів із різних регіонів.

**is_active**: Булеве поле, яке вказує, чи є розсилка активною. Це дозволяє керувати кампаніями та відключати їх, якщо вони більше не актуальні.

**is_general**: Поле, що вказує, чи є ця розсилка загальною або спеціальною. Загальні розсилки можуть бути спрямовані на всю аудиторію, без жорсткого сегментування.

При створенні системи пуш-нотифікацій важливо не тільки налаштувати логіку відправки повідомлень, але й розробити зручний та зрозумілий інтерфейс для їхнього керування. Візуальна частина системи дозволяє користувачам (маркетологам, адміністраторам) легко створювати та редагувати пуш-кампанії, вибирати аудиторії, налаштовувати розклад, а також персоналізувати самі повідомлення. Логіка ж відповідає за точне відправлення пушів у потрібний час або після певної події, забезпечуючи надійність та ефективність розсилок. Тож ось приклад сторінки створення такої пуш-нотифікації.

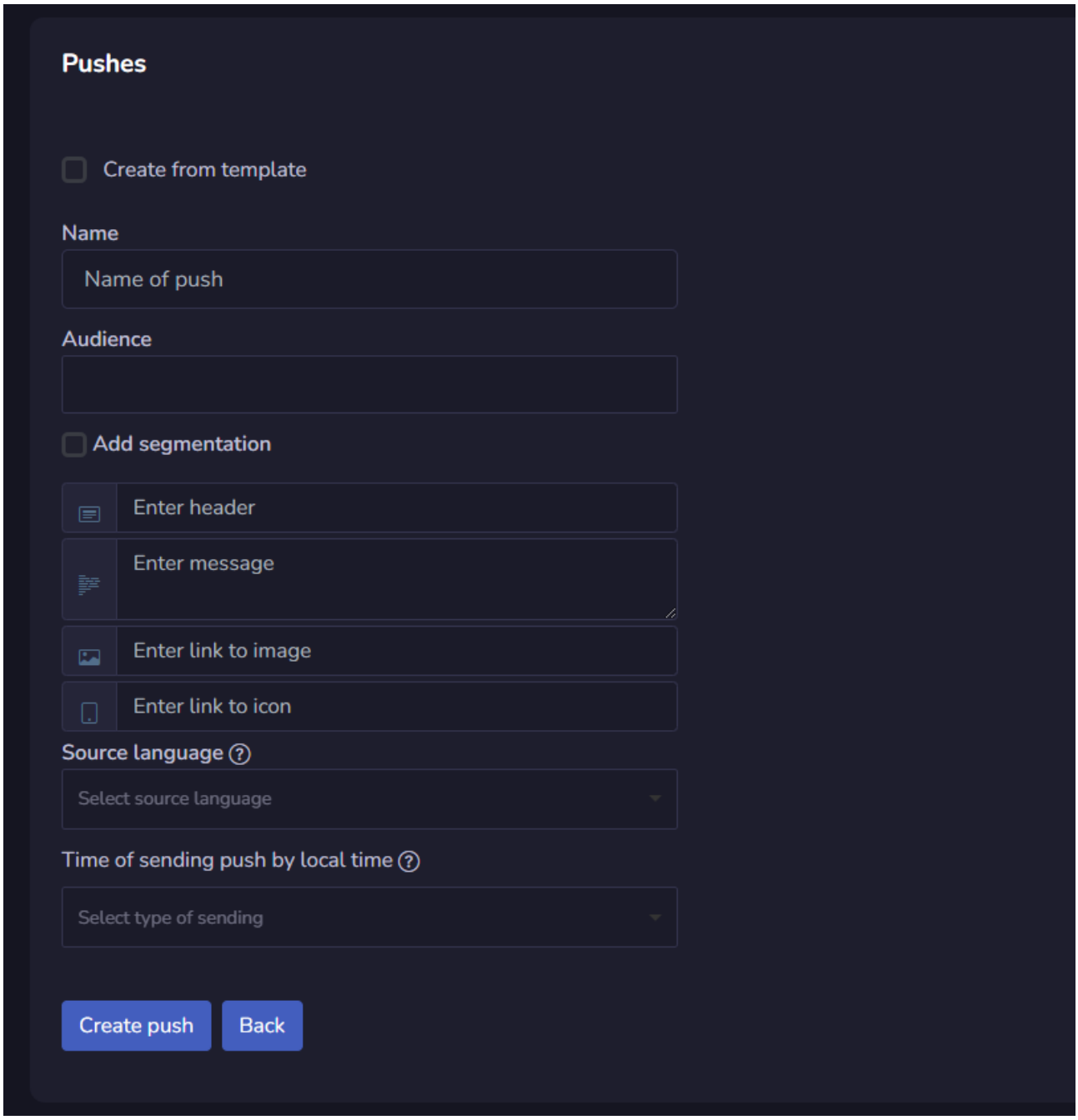



*Рис. 3.9. Сторінка створення пуш-нотифікації*

### 3.6 Розробка функціоналу для створення метрик

Модель **OneSignalNotification** (дивитись у додатку Б) є частиною системи, що інтегрує пуш-нотифікації через платформу OneSignal. Ця модель відповідає за зберігання інформації про відправлені пуші, включаючи деталі про кількість успішних доставок, конверсії та розрахунок клікабельності (CTR). У ній реалізовано кілька важливих методів для отримання статистики та оновлення

даних із сервісу OneSignal. Модель має зв'язки з іншими моделями, такими як **Push**, **MobileApp** і **User**, що дозволяє легко відстежувати взаємодію з різними мобільними додатками та користувачами.

```python
class OneSignalNotification(models.Model):
    push = models.ForeignKey(Push, on_delete=models.CASCADE, related_name='os_notifications', db_index=True)
    app = models.ForeignKey(MobileApp, on_delete=models.CASCADE, related_name='os_notifications')
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='os_notifications', null=True, db_index=True)
    geo = models.CharField(max_length=4, null=True)
    notification_id = models.CharField(max_length=225, db_index=True)
    external_id = models.CharField(max_length=225, null=True, blank=True)
    completed_at = models.DateTimeField(auto_now_add=True, db_index=True)
    updated_at = models.DateTimeField(null=True)
    include_players = models.IntegerField(null=True)
    successful_sent = models.IntegerField(null=True)
    converted = models.IntegerField(null=True)
    ctr = models.IntegerField(null=True)

    def __str__(self):
        return f'{self.push} - {self.notification_id}'

    def _calculate_ctr(self):
        try:
            ctr = round(self.converted / self.successful_sent * 100, 2)
            return ctr
        except ZeroDivisionError:
            return 0

    def get_sent_stat(self):
        failed = self.include_players - self.successful_sent
        return [self.include_players, self.successful_sent, failed]
```

*Рис. 3.10. Реалізація моделі метрик*

```python
def update_data(self, data: dict = None):

    if data is None:
        data = onesignal.get_notification(
            notification_id=self.notification_id,
            app_id=self.app.onesignal_id,
            app_api_key=self.app.api_key
        )

    if data:
        try:
            self.include_players = len(set(data.get('include_player_ids', [])))
            self.successful_sent = data.get('successful', 0)
            self.converted = data.get('converted', 0)
            self.ctr = self._calculate_ctr()
            self.updated_at = datetime.now(timezone.utc)

            return self

        except KeyError as e:
            print(f'KeyError - {str(e)}')
            return {'error': f'Notif ID - {self.id}, KeyError - {str(e)}'}

        except AttributeError as e:
            print(f'AttributeError - {str(e)}')
            return {'error': f'Notif ID - {self.id}, AttributeError - {str(e)}'}
```

*Рис. 3.11. Реалізація моделі метрик*

**push**: Вказує на конкретну пуш-нотифікацію, до якої належить це повідомлення. Зв'язок через ForeignKey з моделлю Push.

**app**: Пов'язує повідомлення з мобільним додатком через ForeignKey до моделі **MobileApp**.

**author**: Вказує на автора, який створив або ініціював повідомлення (може бути null).

**notification_id**: Унікальний ідентифікатор повідомлення у системі OneSignal.

**external_id**: Зовнішній ідентифікатор (необов’язковий), який може бути корисним для додаткових інтеграцій.

**completed_at**: Дата і час завершення або відправки повідомлення.

**updated_at**: Дата останнього оновлення інформації про повідомлення.

**include_players**: Кількість користувачів, яким було надіслано повідомлення.

**geo**: Чотиризначний код, який може вказувати на географічне розташування або регіон.

**successful_sent**: Кількість користувачів, яким повідомлення було успішно доставлено.

**converted**: Кількість користувачів, які взаємодіяли з повідомленням (наприклад, натиснули на нього).

**ctr**: Рівень клікабельності (CTR), що розраховується як відсоток від кількості успішних відправок.

**Методи:**

**_calculate_ctr**: Розраховує CTR (Click-Through Rate) для повідомлення, використовуючи кількість конверсій та успішних відправок. Якщо успішні відправки дорівнюють нулю, повертається 0, щоб уникнути помилки ділення на нуль.

**get_sent_stat**: Повертає статистику відправки повідомлення, включаючи кількість надісланих повідомлень, успішних відправок і кількість невдач (різниця між запланованими та успішними відправками).

**update_data**: Оновлює дані повідомлення, отримані від API OneSignal. Використовується для оновлення кількості включених користувачів, успішних відправок, конверсій та CTR.

Після чого, по аналогії до попередніх розділів, додано логіку для візуалізації потрібних даних.

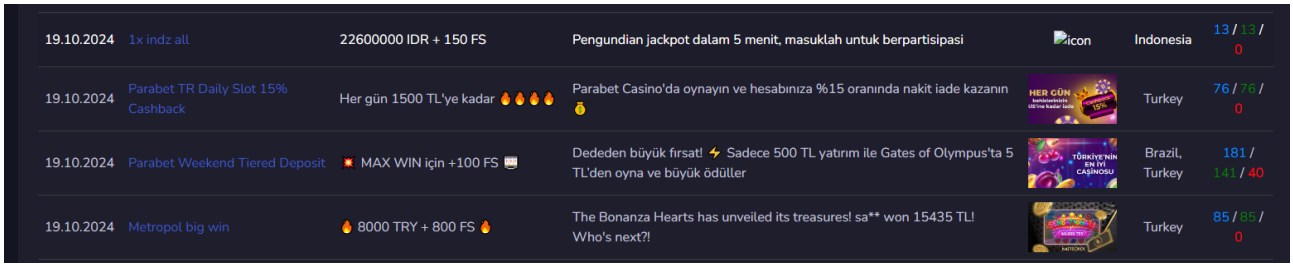



*Рис. 3.12. Сторінка відображення метрик*

### 3.7 Розробка функціоналу для надсилання тестової пуш-нотифікації

У цій в'ющці **SendTestPushView** (дивитись у додатку Б) обробляється логіка відправки тестових пуш-нотифікацій через сторонній сервіс (наприклад, OneSignal). Вона містить два методи: **get** для відображення форми з вибором додатків і мов, та **post** для обробки даних форми та відправки тестового повідомлення.

```python
class SendTestPushView(View):
    def get(self, request):
        if request.GET.get('success'):
            return render(request, 'send-test-push/success.html')

        apps = MobileApp.objects.values_list('id', 'name')
        languages = Language.objects.values_list('code', 'name')

        return render(
            request,
            'send-test-push/send-test-push.html',
            {
                'apps': apps,
                'push_languages': languages
            }
        )

    def post(self, request):
        data = request.POST.copy()

        app_id = data.get('app')
        user_ip = data.get('user_ip')
        header = data.get('header')
        message = data.get('message')
        image_url = data.get('image_url')
        icon_url = data.get('icon_url')
        language = data.get('language')

        app = MobileApp.objects.get(pk=app_id)

        # Get external_user_id from KT
        try:
            kt = WildWildKtDash()
            external_user_id = kt.get_external_id_by_ip(user_ip=user_ip, app_bundle=app.bundle_name)
        except IndexError:
            messages.error(request, _('There is no device with this IP'))
            return JsonResponse({'success': False, 'error': 'Bad IP address'}, status=400)

        # Get user data from OS
        user_data = onesignal.get_user_data_by_external_id(
            app_id=app.onesignal_id,
            app_api_key=app.api_key,
            external_id=external_user_id
        )

        # Send test push
        translator = Translator()
        header = translator.translate(header, language)
        message = translator.translate(message, language)

        onesignal.test_push(
            app, user_data.player_id,
            image=image_url,
            icon=icon_url,
            header=header,
            message=message
        )

        return JsonResponse({'success': True})
```

*Рис. 3.12. Реалізація в'юшки для надсилання тестової пуш-нотифікації*

**Метод get**:

- Якщо параметр success надано у GET-запиті, користувач перенаправляється на сторінку успіху після відправки тестової пуш-нотифікації.
- Отримуються списки доступних мобільних додатків та мов, щоб їх можна було відобразити у формі для відправки пуш-нотифікацій.
- Ці дані передаються у контекст і рендеряться у шаблоні.

**Метод post**:

- Спочатку витягуються всі дані з POST-запиту, такі як ID додатку, IP користувача, заголовок повідомлення, текст, URL зображень, та мова.
- Далі вибраний мобільний додаток отримується за допомогою його ID з бази даних.
- За допомогою API-сервісу (наприклад, WildWildKtDash), на основі IP-адреси отримується **external_user_id**. Якщо IP некоректний, повертається помилка.
- На основі цього **external_user_id**, за допомогою OneSignal API, отримується інформація про користувача.
- Використовуючи бібліотеку перекладу (наприклад, Translator), заголовок і текст повідомлення перекладаються на вибрану мову.
- Викликається метод test_push, який відправляє пуш-нотифікацію конкретному користувачу (ідентифікованому за **player_id**) із вказаними даними.

Ця в'юшка забезпечує зручний інтерфейс для тестування пуш-нотифікацій, дозволяючи обирати різні додатки та мови, а також використовує API для автоматизації пошуку користувачів за IP і відправки повідомлень.

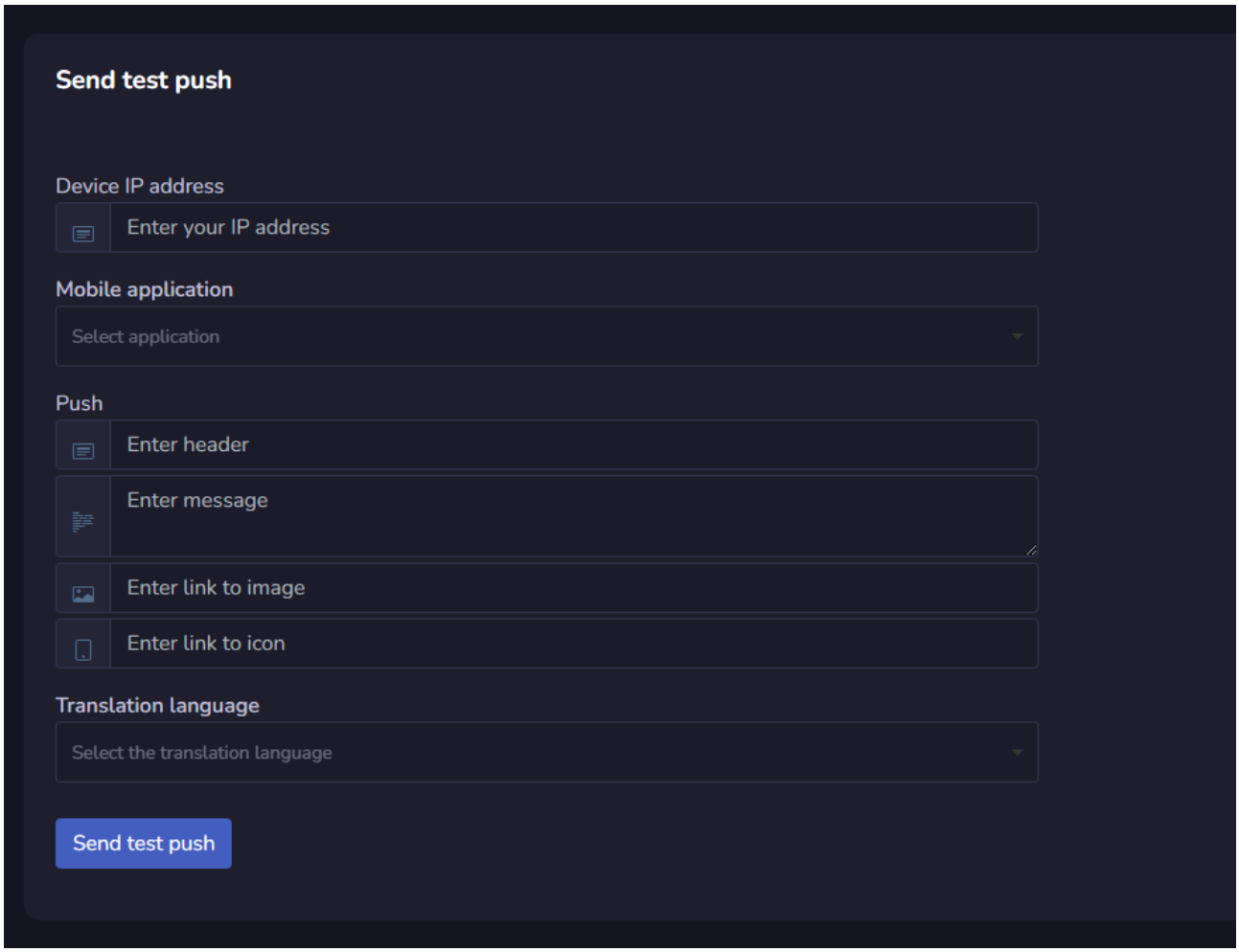



*Рис. 3.13. Сторінка створення тестової пуш-нотифікації*

# 4 Інструкція користувачу

***Авторизація***

Спочатку користувача зустрічає сторінка авторизації у систему

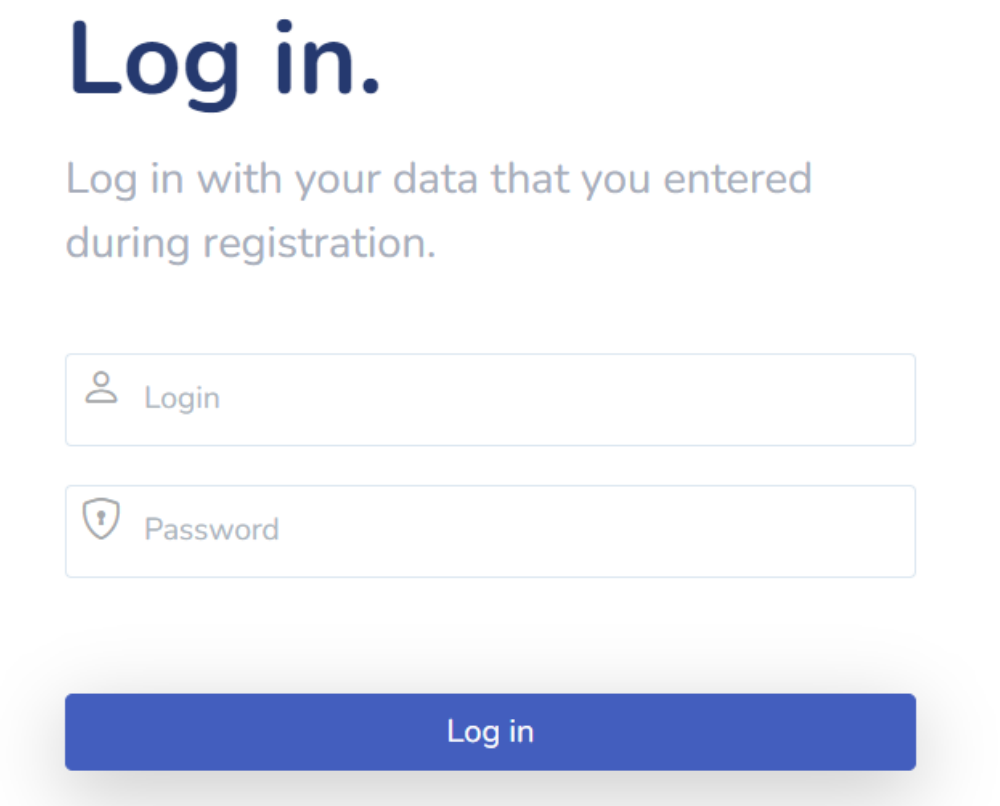



*Рис. 4.1. Сторінка авторизації*

Кожен користувач отримує доступи від адміністратора, тож можливості власноруч зареєструватися немає.

***Загальний дешборд***

Після успішної авторизації, користувач потрапляє на головну сторінку, де наводиться загальна статистика по його трафіку.


*Рис. 4.2. Дешборд статистики користувача згідно з його трафіком*

Рисунок 4.2 представляє **панель управління активною аудиторією** у системі аналітики, яка відстежує важливі показники користувацької активності для мобільного додатку. Панель призначена для відображення даних про встановлення додатка, реєстрації та депозити користувачів, а також для візуалізації статистики за обраний період часу.

**Ключові елементи панелі:**

**Вибір фільтрів:**

- У верхній частині екрану є фільтри для вибору додатка (All Apps) та географічного розташування (All Geo). Це дозволяє користувачам панелі фільтрувати дані за конкретним додатком або регіоном, щоб отримати більш детальну аналітику для окремих сегментів користувачів.

**Загальні показники**:

- **Active audience (Активна аудиторія)**: Відображає кількість активних користувачів, яка на даний момент становить 975.
- **Installs (Встановлення)**: Вказує на кількість інсталяцій додатку, що дорівнює 660.
- **Registrations (Реєстрації)**: Показує кількість реєстрацій користувачів, яка становить 247.
- **Deposits (Депозити)**: Відображає кількість депозитів, зроблених користувачами, і на момент скріншота вона становить 68.

**Секція "Registrations by push" та "Deposits by push"**:

- Ці поля показують кількість реєстрацій і депозитів, які були здійснені безпосередньо через пуш-нотифікації. На скріншоті видно, що реєстрації та депозити через пуш-нотифікації на даний момент дорівнюють 0, що може вказувати на відсутність або неефективність кампанії пушів у цей період.

**Графік "Statistics of installs for month"**:

- Графік відображає статистику встановлень додатку за місяць. Вісь X представляє дні місяця, а вісь Y – кількість інсталяцій. З графіка видно, що активність користувачів коливається, з найбільшим піком на 23-й день місяця, коли було зареєстровано 7 інсталяцій. Після цього активність знову знижується.

  Ліворуч на сторінці присутня бокова панель.

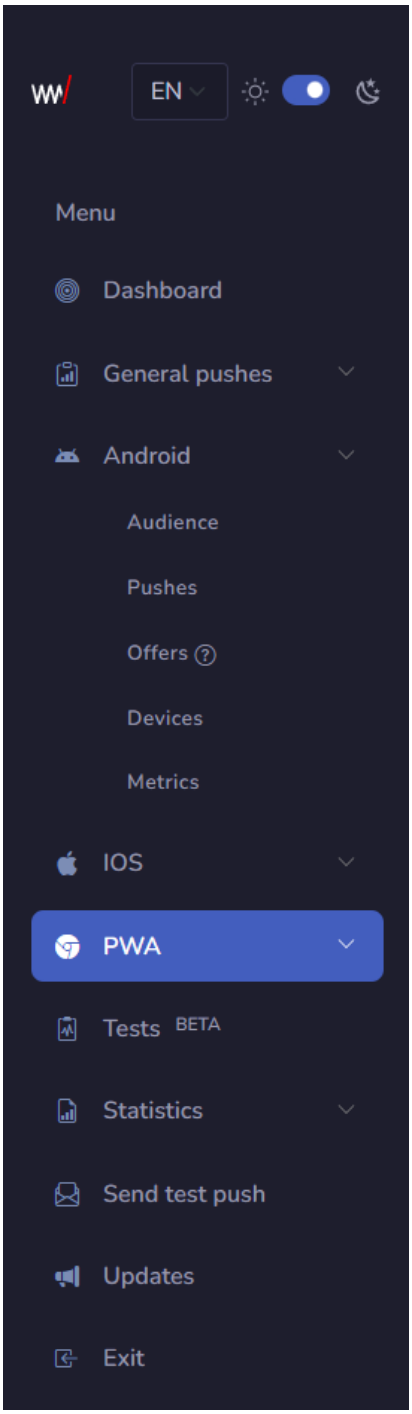



*Рис. 4.3. Бокова панель*

Вона представляє собою такі поля:

- **Dashboard** – дає можливість подивитися статистику за додатками у вигляді діаграм.
- **General pushes** – можливість створювати загальну розсилку для вибраних користувачів

- **Android, IOS, PWA** мають у собі наступне:

    1. **Audience** – розділ, у якому ви можете створювати та редагувати вже створені аудиторії.

    2. **Pushes** – це розділ, де ви можете додавати нові пуш-нотифікації для аудиторій.

    3. **Offers** – у цьому розділі ми додаємо назву офферів.

    4. **Devices** – розділ, у якому ви можете чекнути активні та неактивні пристрої користувачів.

    5. **Metrics** – розділ зі статистикою. Наразі в ньому відображаються дані по всіх відправлених пуш-нотифікаціях.

- **Tests** – можливість створення шаблонів тестовихпуш-нотифікацій;

- **Статистика** - ця опція дає можливість переглянути стастику по клієнтам і самим додаткам, на котрі здійснюються пуш-нотифікації;

- **Send test push** – можливість надсилання тестової пуш-нотифікації

- **Updates** – інформацію про оновлення у системі;

- **Exit** – вихід з особистого кабінету

Продемонструємо роботу панелі на прикладі Android. Для решти ОС дії аналогічні.

***Створення оффера***

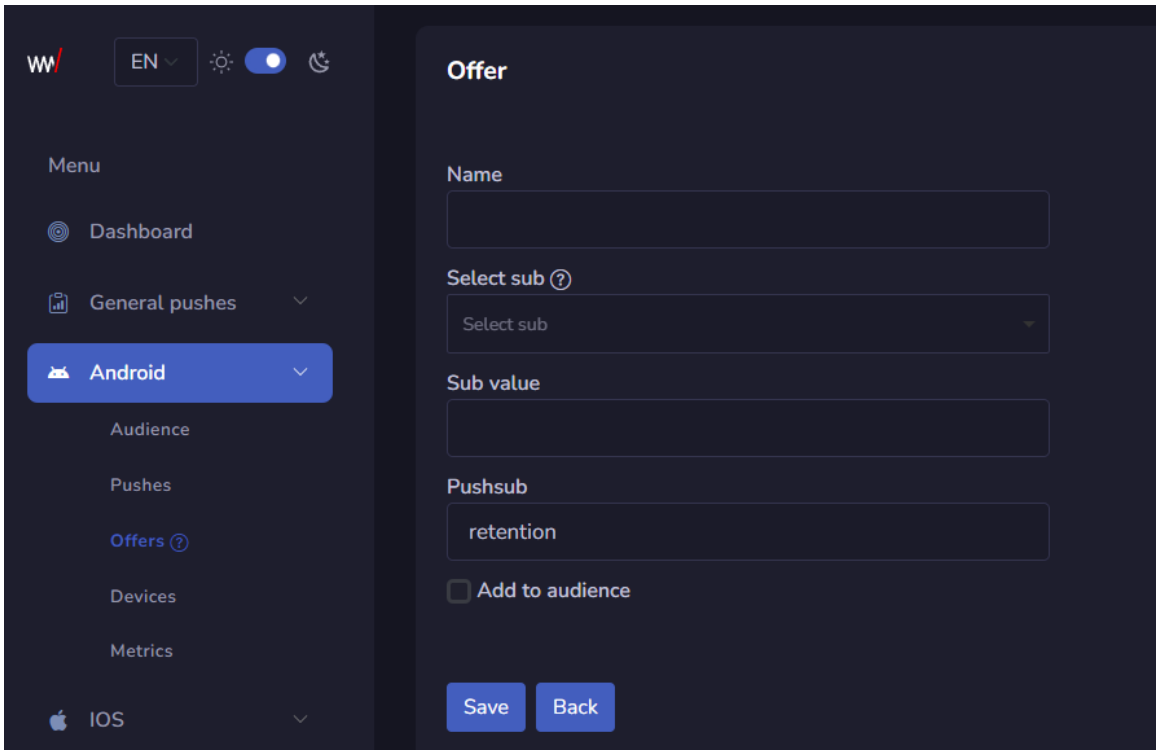



*Рис. 4.4. Форма створення оффера*

У полі «Name» прописуємо назву оффера за принципом «Name| GEO». Важливо прописувати саме за такою системою, щоб надалі було легко додавати та видаляти оффери з аудиторії.

У полі «Select SUB» – потрібно вибрати одне із значення. Усього є sub2, sub3, sub4, sub5, sub6. Кожен sub – це можливість встановити унікальний ідентифікатор під час запуску трафіку. Якщо для одного оффера використовувати кілька варіантів тесту, тоді потрібно проходити етап «Створити оффер» та прописувати sub2, в іншому sub3 тощо.

Поле «Sub value» – це і є той самий ідентифікатор, прописаний у засланні. Якщо за оффером не планується спліт тестів, тоді в цьому полі прописується просто 1.

Поле «Pushsub» – це ключ команди. Після заповнення усіх полів потрібно натиснути «Save».

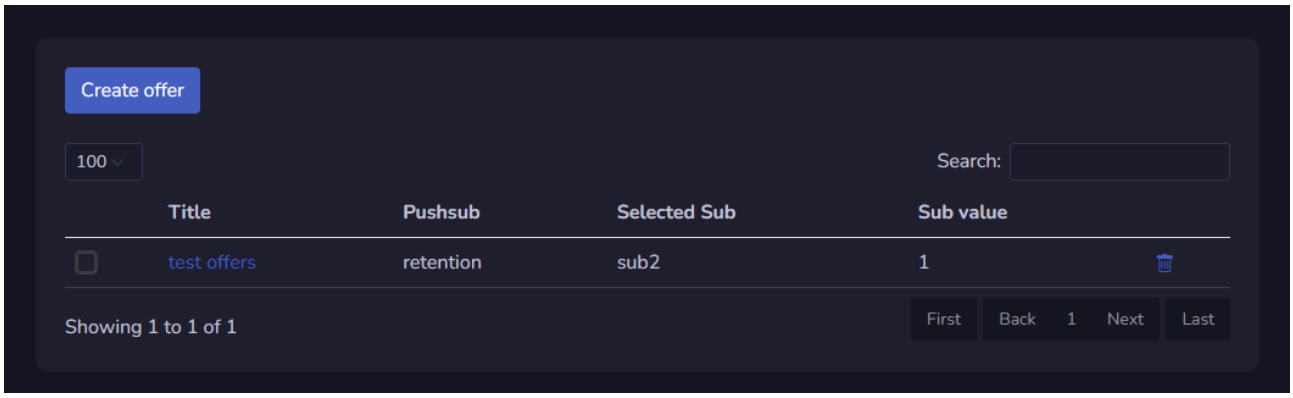



*Рис. 4.5. Таблиця створених офферів*

***Створення аудиторії***

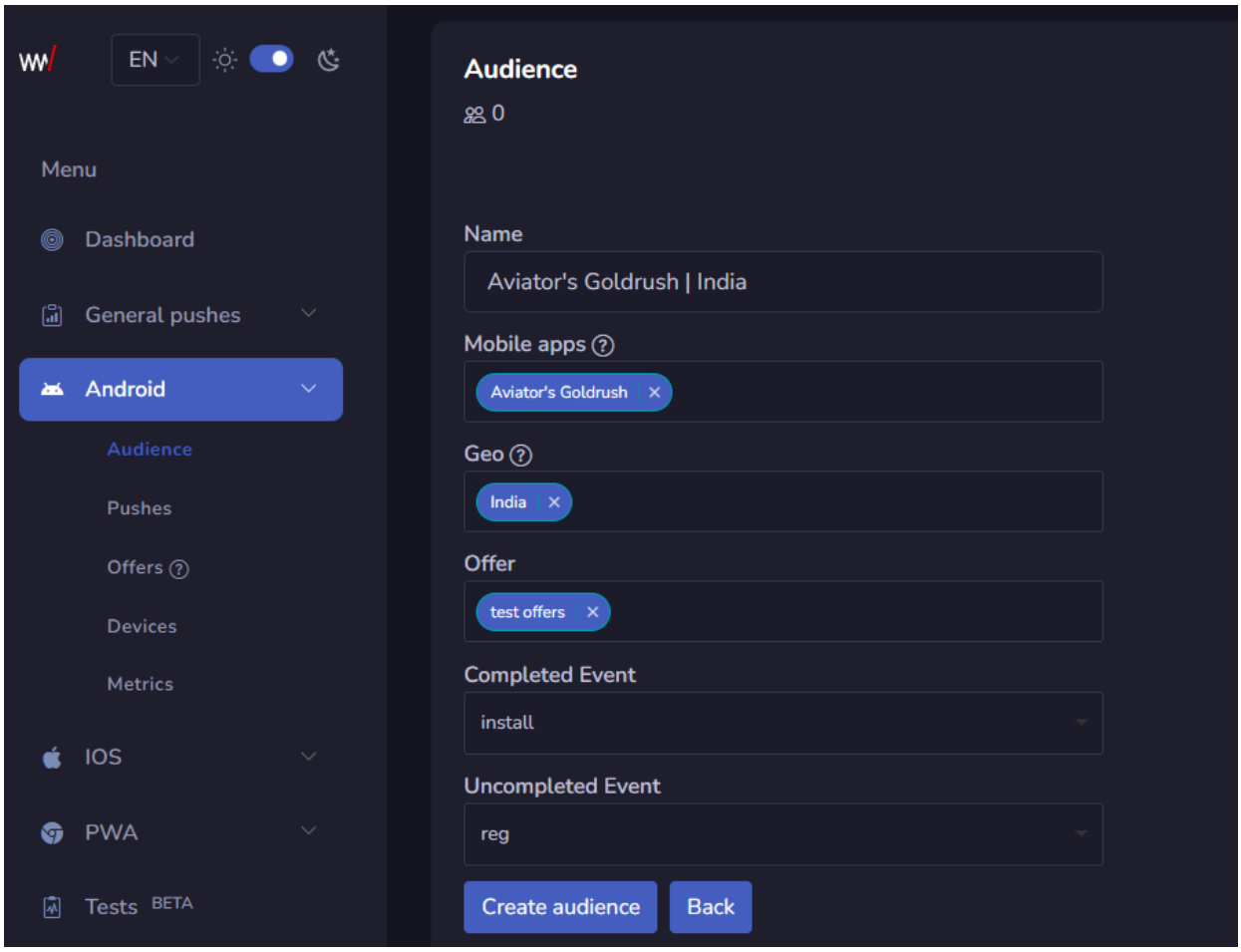



*Рис. 4.6. Форма створення аудиторії*

Після додавання оффера (або офферів) потрібно перейти у вкладку «Audience».

**Name** – назва аудиторії

**Mobille apps** – вибрати додатки, на які повинна бути розсилка

**Geo** – вибрати країни, які повинна охопити розсилка

**Offer** – щойно створений оффер

**Completed Event** – тобто завершена подія, розіслати тим користувачам, які наприклад встановили додаток

**Uncompleted event** – та сама логіка, проте тут можна вказати тих користувачів, які могли зареєструватися на певному сервісі, проте не встановили їх додаток

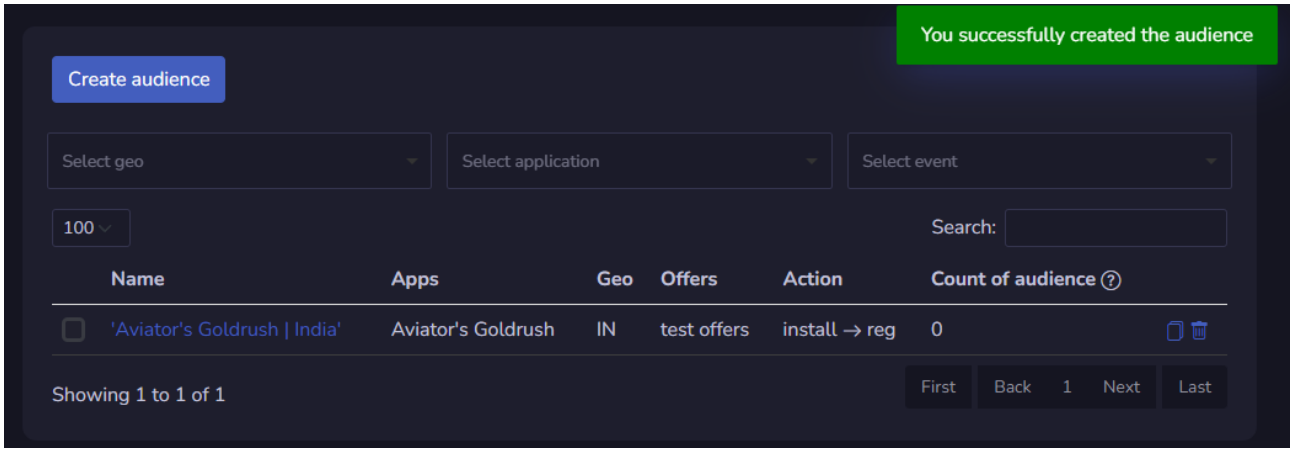



*Рис. 4.7. Таблиця створених аудиторій*

***Створення пуш-нотифікації***

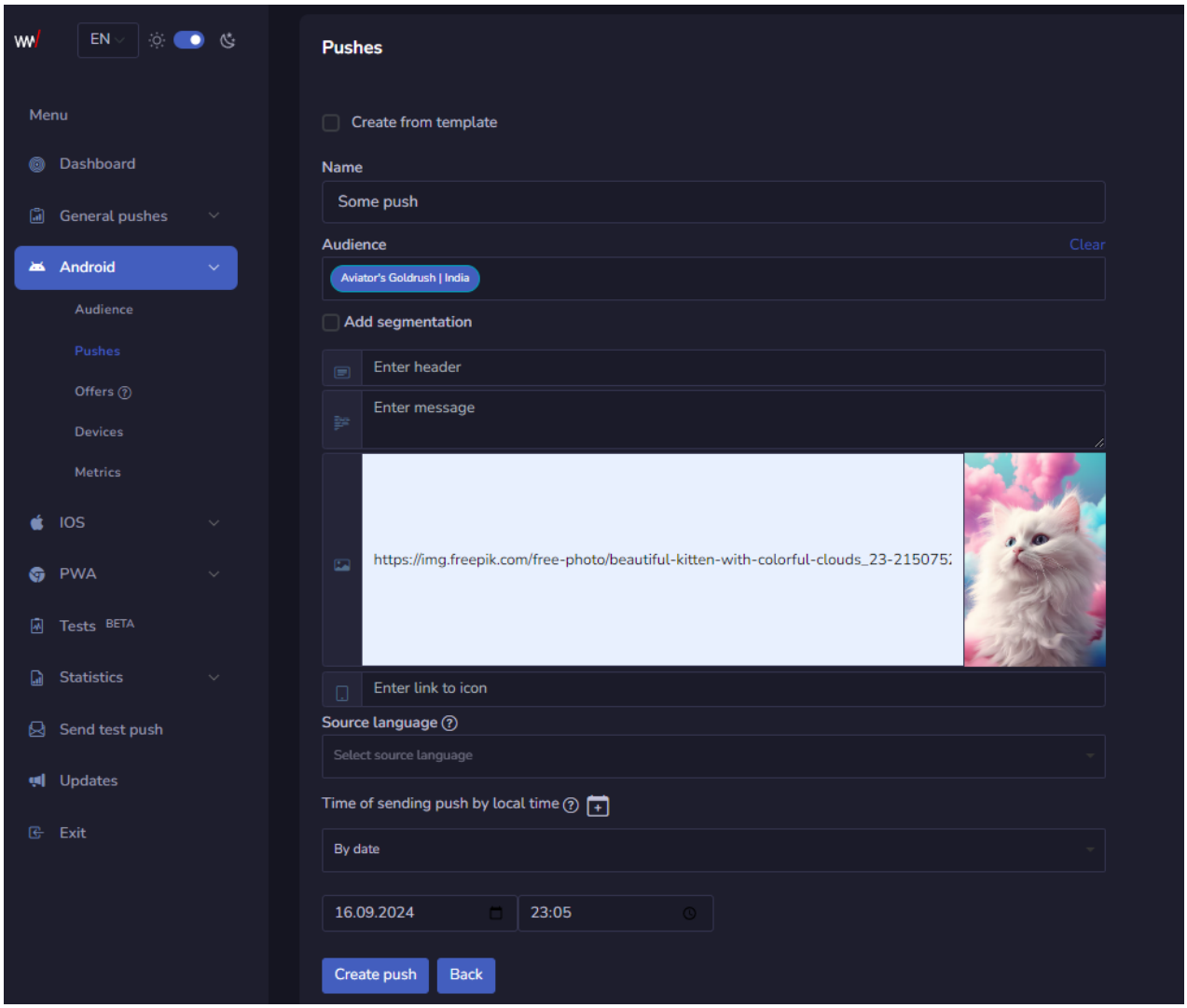



*Рис. 4.7. Форма створення пуш-нотифікації*

**Name** – назва пуш-нотифікації

**Audience** – можна обрати одну або декілька створених аудиторій

**Add segmentation** – інструмент з фільтрами всередині аудиторії «остання активність, реєстрація/інстал або ні»

Відповідні поля – це поля під заголовок пуша, його тіло (повідомлення), посилання на картинку, яка прийде на мобільний телефон, і посилання на іконку.

**Source language** – тут можна вибрати мову і API перекладача згенерує пуш-нотифікацію вибраною мовою

І останнє, можна вказати конкретну годину надсилання пуш-нотифікації або певний інтервал

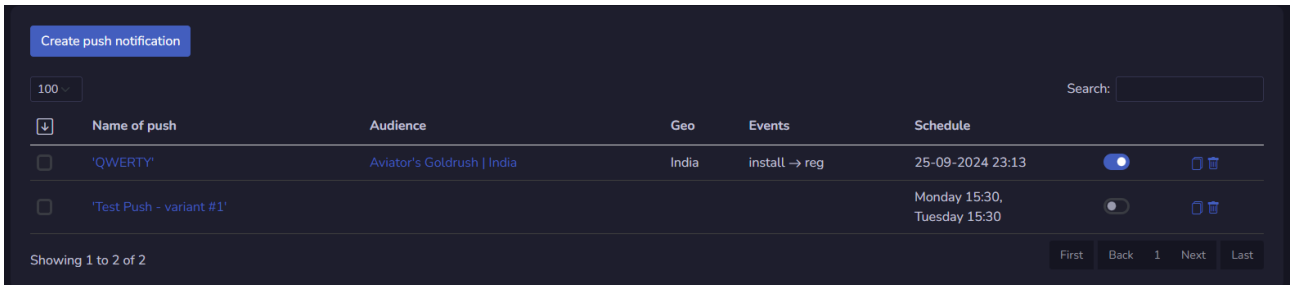



*Рис. 4.8. Таблиця з доступними пуш-нотифікаціями*

***Створення тестової пуш-нотифікації***

Користувачу потрібно заповнити поля, а саме:

- Device IP address: IP адреса пристрою, на який він бажає здійснити розсилку
- Mobile application: вибрати один додаток або декілька
- Перші два поля у розділі **Push** відповідають за заголовок пуш-нотифікації так текст повідомлення
- Два останніх поля відповідають за посилання на головну картинку пуш-нотифікації та іконку
- Translation Language: користувач обирає мову, на котру потрібно перекласти пуш-нотифікацію

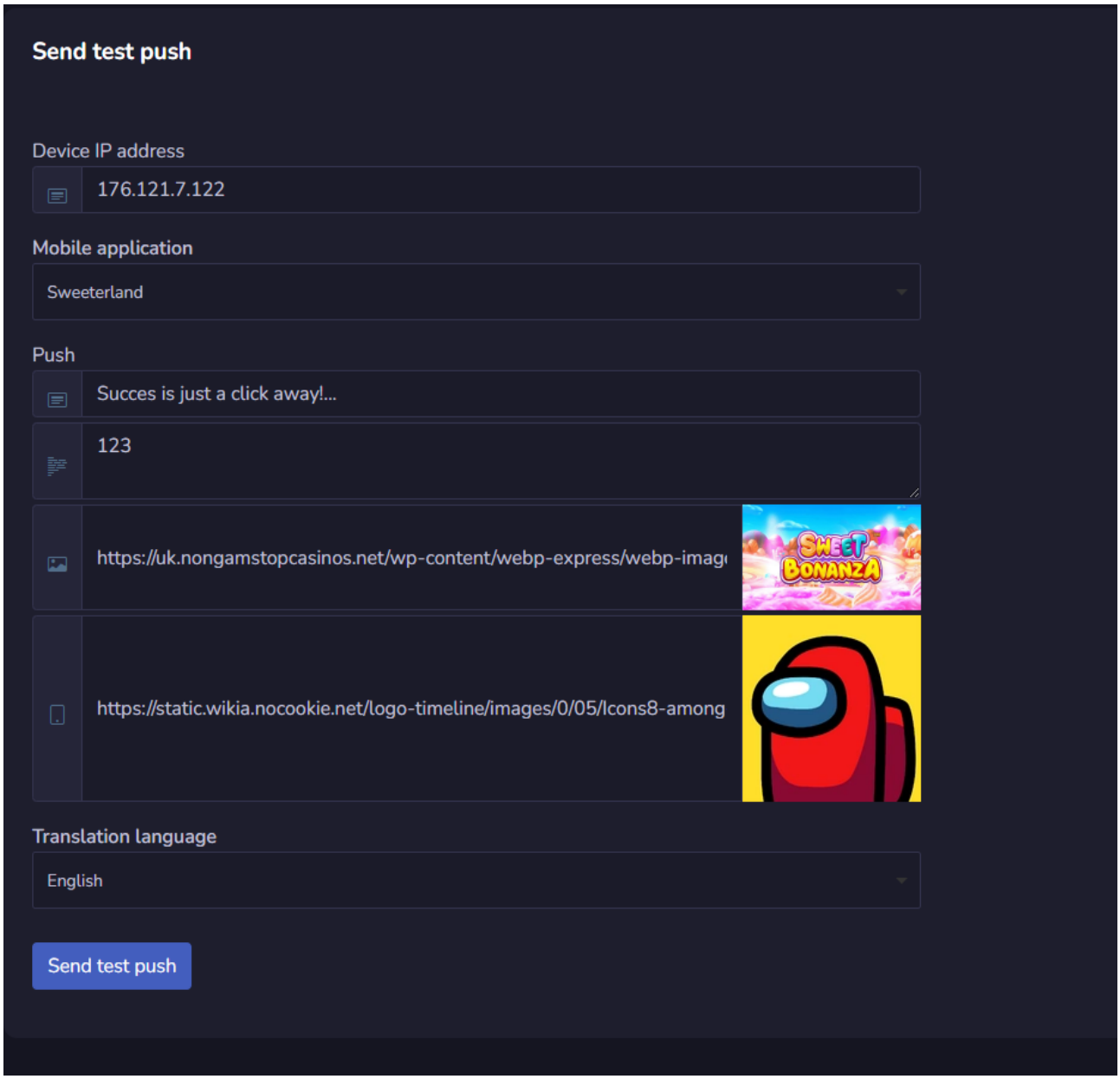



*Рис. 4.9. Заповнення форми тестового пуша*

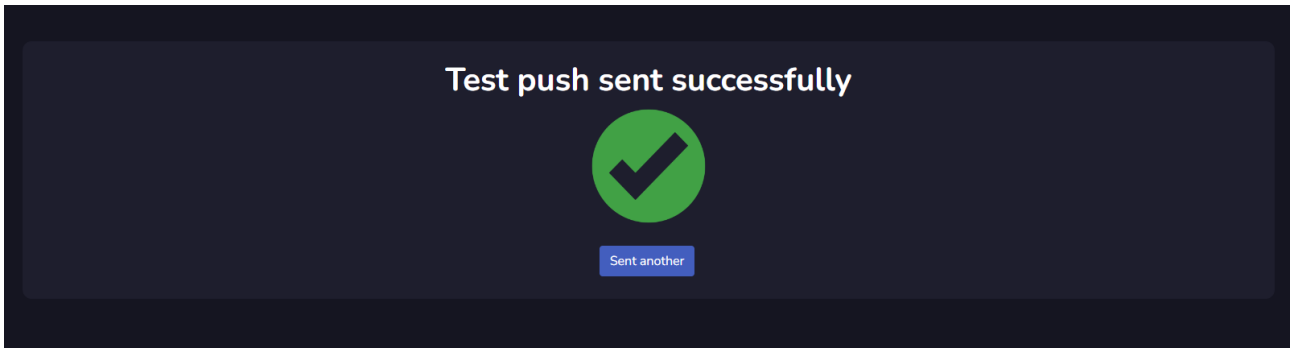



*Рис. 4.10. Підтвердження форми, що все пройшло успішно*

На мобільному пристрої пуш-нотифікація набуває вигляду

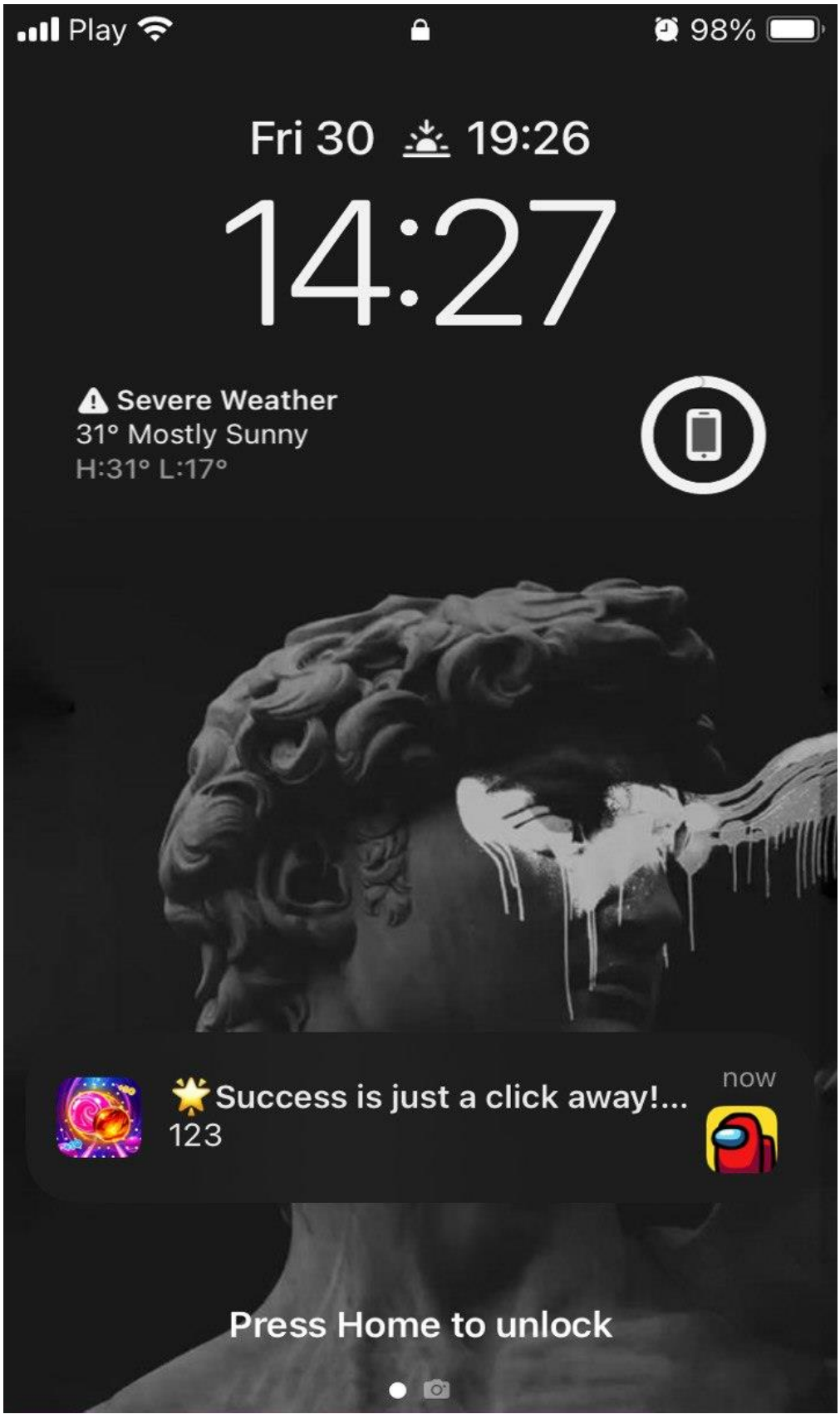



*Рис. 4.11. Пуш-нотифікація на мобільному пристрої*

## Висновки

Отже, створена пуш-система надає користувачам можливість ефективно управляти push-нотифікаціями для залучення аудиторій. Інтерфейс дозволяє легко створювати та налаштовувати push-нотифікації, відстежувати метрики результативності та аналізувати відгук аудиторії на повідомлення. Інтеграція офферів розширює можливості для маркетингових кампаній, дозволяючи користувачам швидко та ефективно вибирати цільові пропозиції для відповідних сегментів аудиторії.

Можливість відправляти тестові повідомлення підвищує якість взаємодії з аудиторією, забезпечуючи перевірку правильності відображення і налаштувань push-повідомлень перед запуском.

Проєкт підкреслює мою здатність вирішувати актуальні завдання у сфері розробки, а також важливість якісного планування, зосередженості на користувацькому досвіді та знань у створенні масштабованих рішень для бізнесу.

Ця робота надала мені цінний досвід, допомогла чітко визначити професійну мету, структурувати процеси та використовувати новітні підходи у створенні ефективних інструментів для бізнесу.

## Перелік використаних джерел

1. One Signal [Електронний ресурс]. – URL : https://onesignal.com/

2. Firebase Cloud Messaging [Електронний ресурс]. – URL : https://firebase.google.com/docs/cloud-messaging?hl=en

3. Leanplum [Електронний ресурс]. – URL : https://docs.leanplum.com/reference/android-setup

4. Python [Електронний ресурс]. – URL : https://foxminded.ua/de-vykorystovuietsia-python/#:~:text=Python%20%E2%80%93%20%D1%86%D0%B5%20%D0%B2%D0%B8%D1%81%D0%BE%D0%BA%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D0%B5%D0%B2%D0%B0%20%D0%BC%D0%BE%D0%B2%D0%B0%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F,%D1%80%D1%96%D1%88%D0%B5%D0%BD%D1%8C%20%D1%80%D1%96%D0%B7%D0%BD%D0%BE%D0%B3%D0%BE%20%D0%BC%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D1%83%20%D1%82%D0%B0%20%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F.

5. Django [Електронний ресурс]. – URL : https://wezom.com.ua/ua/blog/django-ili-laravel

6. Celery [Електронний ресурс]. – URL : https://evergreens.com.ua/ua/articles/celery-flower-queues.html

7. Redis [Електронний ресурс]. – URL : https://www.ukraine.com.ua/wiki/redis/overview/

8. PostgreSQL [Електронний ресурс]. – URL : https://foxminded.ua/postgresql-shcho-tse/

9. HTML [Електронний ресурс]. – URL : https://goit.global/ua/articles/html-i-css-shcho-tse-komu-ta-dlia-choho-potribno/

10. CSS [Електронний ресурс]. – URL : https://avada-media.ua/css/

11. JavaScript [Електронний ресурс]. – URL : https://goit.global/ua/articles/shcho-take-javascript-i-dlia-choho-vin-potriben/

12. Docker [Електронний ресурс]. – URL : https://spacelab.ua/articles/Docker/

# Додатки

## Додаток А

```
from django.db import models
from django.contrib.auth.models import User

from apps.mobileapps.models import Platforms

EXCLUSIONS_PUSHSUBS = ('foxes',)

class Offer(models.Model):
    name = models.CharField(max_length=255)
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='offers', null=True)
    offer_id = models.IntegerField(null=True, blank=True)
    selected_sub = models.CharField(max_length=255, null=True, blank=True)
    sub_value = models.CharField(max_length=255, null=True, blank=True)
    push_sub = models.CharField(max_length=255, null=True, blank=True)
    device_type = models.IntegerField(choices=Platforms.choices, null=True, blank=True)

    def __str__(self):
        return self.name

    @staticmethod
    def get_ignore_values(pushsubs):
        if any(any(exc_pushsub in pushsub for exc_pushsub in EXCLUSIONS_PUSHSUBS) for pushsub in pushsubs):
            return ('None',)
        return '1', 'None'

    def is_valid(self):
        pushsubs = self.author.groups.values_list('name', flat=True)
        return self.selected_sub and self.sub_value not in self.get_ignore_values(pushsubs)


class Audience(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(User, on_delete=models.CASCADE, db_index=True)
    device_type = models.IntegerField(choices=Platforms.choices, null=True, blank=True, db_index=True)
    apps_type = models.CharField(max_length=50, choices=AppType.choices, default=AppType.GAMBLING, blank=True,
db_index=True)
    geo = models.ManyToManyField(Geo)
    apps = models.ManyToManyField(MobileApp)
    all_applications = models.BooleanField(default=False, blank=True)
    all_geos = models.BooleanField(default=False, blank=True)
    all_push_subs = models.BooleanField(default=False, blank=True)
    push_subs = models.ManyToManyField(Group, blank=True)
    offers = models.ManyToManyField(Offer, blank=True)
    filter_by_offer = models.BooleanField(default=True)
    finished_event = models.CharField(max_length=30, choices=UserEvent.choices, null=True, blank=True)
    unfinished_event = models.CharField(max_length=30, choices=UserEvent.choices, blank=True)
    clicked_push_counter = models.IntegerField(default=0)
    deleted_app = models.IntegerField(default=0)
    received_push_counter = models.IntegerField(default=0)
    users_count = models.PositiveIntegerField(default=0)
    deposit_counter = models.IntegerField(default=0, blank=True)
    is_general = models.BooleanField(default=False, blank=True)

    def __str__(self):
        return f'{self.name}'

    def select_all_apps(self):
        self.apps.set(MobileApp.objects.all())

    @staticmethod
    def get_filters(mobileapps_data, geos, pushsub, offers, finished_event, unfinished_event, device_type):
        if 'all' in mobileapps_data:
            mobileapps = MobileApp.objects.all().values_list('pk', flat=True)
        else:
```

```
            mobileapps = MobileApp.objects.filter(pk__in=mobileapps_data).values_list('pk', flat=True)

        q_filters = [Q(installed_status=True)]
        if mobileapps:
            q_filters.append(Q(mobile_app__pk__in=mobileapps))
        if pushsub:
            pushsub = Group.objects.filter(pk__in=pushsub).values_list('name', flat=True)
            q_filters.append(Q(pushsub__in=pushsub))
        if geos:
            geos = Geo.objects.filter(pk__in=geos).values_list('code', flat=True)
            q_filters.append(Q(geo_code__in=geos))
        if offers:
            offers = Offer.objects.filter(pk__in=offers)
            if 'wwlpush' in pushsub:
                offers = offers.values_list('offer_id', flat=True)
                q_filters.append(Q(offer_id__in=offers))
            else:
                offers = offers.exclude(sub_value__in=Offer.get_ignore_values(pushsub))
                if offers.exists():
                    subs = [
                        Q(**{offer.selected_sub: offer.sub_value}) for offer in offers
                        if offer.is_valid()
                    ]
                    if subs:
                        q_filters.append(reduce(or_, subs))
        if device_type is not None:
            q_filters.append(Q(device_type=device_type))
        if finished_event:
            q_filters.append(Q(finished_event=finished_event))
        if unfinished_event:
            q_filters.append(Q(unfinished_event=unfinished_event))

        return q_filters

    @staticmethod
    def calculate_audience_users_count(*args, **kwargs):
        filters = Audience.get_filters(*args, **kwargs)

        return AppUser.objects.filter(*filters).values_list('external_user_id', flat=True).distinct().count()

    def get_audience_users(self, read_only=True):
        filters = Audience.get_filters(
            self.apps.values_list('pk', flat=True),
            self.geo.values_list('pk', flat=True),
            self.push_subs.values_list('pk', flat=True),
            self.offers.values_list('pk', flat=True),
            self.finished_event,
            self.unfinished_event,
            self.device_type
        )

        if read_only:
            return AppUser.objects.using('readonly').filter(*filters)
        return AppUser.objects.filter(*filters)

    def get_audience_users_count(self):
        return self.get_audience_users().values_list('external_user_id', flat=True).distinct().count()


class AudienceSegmentaion(models.Model):
    class SegmentType:

        LAST_ACTIVE = 'last_active'
        EVENT_DELAY = 'event_delay'
        IS_DEP = 'is_dep'

        choices = (
            (LAST_ACTIVE, _('Last activity')),
            (EVENT_DELAY, _('Days after action')),
```

```
            (IS_DEP, _('A deposit has been made'))
        )

    class ComparsionType:
        MORE = 'lte'
        LESS = 'gte'
        TRUE = 'True'
        FALSE = 'False'

        choices = (
            (MORE, 'More'),
            (LESS, 'Less'),
            (TRUE, 'Yes'),
            (FALSE, 'No')
        )

    class EventType:
        INSTALL = 'install'
        REG = 'reg'
        DEP = 'dep'

        choices = (
            (INSTALL, 'Install'),
            (REG, 'Registration'),
            (DEP, 'Deposit')
        )

    segment = models.CharField(max_length=150, choices=SegmentType.choices)
    event = models.CharField(max_length=150, choices=EventType.choices, null=True, blank=True)
    comparison = models.CharField(max_length=50, choices=ComparsionType.choices)
    time_period = models.PositiveIntegerField(default=0, null=True, blank=True)

    def __str__(self):
        return (f'{self.segment} - {self.event or ""} {self.comparison} '
                f'{(str(self.time_period) + " days") if self.time_period else ""}')

    def filter_by_segment(self, appuser_query):
        if self.segment == self.SegmentType.IS_DEP:
            return appuser_query.filter(is_dep=self.comparison)

        time_range = timezone.now() - timedelta(days=self.time_period)

        if self.segment == self.SegmentType.LAST_ACTIVE:
            lookup = f'last_active__{self.comparison}'
            return appuser_query.filter(**{lookup: time_range})

        if self.segment == self.SegmentType.EVENT_DELAY:
            if self.event in (self.EventType.INSTALL, self.EventType.REG):
                user_fields = {
                    self.EventType.INSTALL: 'install_date',
                    self.EventType.REG: 'registration_date'
                }
                lookup = f'{user_fields.get(self.event)}__{self.comparison}'
                return appuser_query.filter(**{lookup: time_range})
            elif self.event == self.EventType.DEP:
                lookup = f'last_deposit__{self.comparison}'
                return appuser_query.annotate(last_deposit=Max('deps__created')).filter(**{lookup: time_range})

    @staticmethod
    def get_comparison(segment_type):
        if segment_type in (AudienceSegmentaion.SegmentType.LAST_ACTIVE,
AudienceSegmentaion.SegmentType.EVENT_DELAY):
            return (
                (AudienceSegmentaion.ComparsionType.MORE, 'More'),
                (AudienceSegmentaion.ComparsionType.LESS, 'Less')
            )
        elif segment_type == AudienceSegmentaion.SegmentType.IS_DEP:
            return (
                (AudienceSegmentaion.ComparsionType.TRUE, 'Yes'),
```

```
            (AudienceSegmentaion.ComparsionType.FALSE, 'No')
        )

    @staticmethod
    def get_events():
        return AudienceSegmentaion.EventType.choices

    def get_self_comparison(self):
        return self.get_comparison(self.segment)
```

## Додаток Б

```
class Push(models.Model):
    MAILING_DATE = 'date'
    MAILING_DAY = 'day'
    EVENT_PUSH = 'event'

    MAILING_TYPE = (
        (MAILING_DAY, 'Mailing by day'),
        (MAILING_DATE, 'Mailing by date'),
        (EVENT_PUSH, 'Mailing by event')
    )

    title = models.CharField(max_length=150)
    author = models.ForeignKey(User, on_delete=models.CASCADE, db_index=True)
    device_type = models.IntegerField(choices=Platforms.choices, null=True, blank=True)
    audiences = models.ManyToManyField(to=Audience)
    segmentations = models.ManyToManyField(to=AudienceSegmentaion, blank=True)
    mailing_schedule = models.ManyToManyField(to=MailingSchedule, blank=True)
    mailing_date = models.ManyToManyField(to=MailingDate, blank=True)
    event_push = models.ForeignKey(to=EventPush, on_delete=models.SET_NULL, null=True, blank=True)
    header = models.CharField(max_length=150)
    message = models.TextField()
    img_url = models.URLField(max_length=500, null=True, blank=True)
    icon_url = models.URLField(max_length=500, null=True, blank=True)
    endpoint_url = models.URLField(max_length=500, null=True, blank=True)
    push_tasks = models.ManyToManyField(PeriodicTask)
    source_language = models.CharField(max_length=225, null=True)
    is_active = models.BooleanField(default=True)
    is_general = models.BooleanField(default=False)

    def __str__(self):
        return f'{self.header}'

    def set_active_state(self, enable):
        self.is_active = enable
        self.save()

    def update_tasks_state(self):
        tasks_to_update = self.push_tasks.all()
        for task in tasks_to_update:
            task.enabled = self.is_active
        PeriodicTask.objects.bulk_update(tasks_to_update, ['enabled'])

    def calculate_ctr(self, by_author=None):
        try:
            notifications = self.os_notifications
            if by_author is not None:
                notifications = notifications.filter(author=by_author)

            notifications_convert = sum(notifications.values_list('converted', flat=True))
            notifications_sents = sum(notifications.values_list('successful_sent', flat=True))
            return round((notifications_convert / notifications_sents) * 100, 2)
        except ZeroDivisionError:
            return 0

    def get_last_notifications_date(self):
```

```
    last_notification = self.os_notifications.last()

    if not last_notification:
        return '-'
    return last_notification.completed_at.strftime('%d.%m.%Y %H:%M:%S')

def get_clicks(self, by_author=None):
    notifications = self.os_notifications.all()
    if by_author is not None:
        notifications = notifications.filter(author=by_author)

    return sum(click for click in notifications.values_list('converted', flat=True) if click)

def get_sents_stat(self, by_author=None):
    notifications = self.os_notifications.all()
    if by_author is not None:
        notifications = notifications.filter(author=by_author)

    total = sum([notification.include_players for notification in notifications if notification.include_players])
    sent = sum([notification.successful_sent for notification in notifications if notification.successful_sent])
    failed = total - sent
    return [total, sent, failed]

def generate_crontab_task(self, mailing_date, timezone, timestamp):
    try:
        schedule, _ = CrontabSchedule.objects.get_or_create(
            minute=mailing_date.time.minute,
            hour=mailing_date.time.hour,
            day_of_week=mailing_date.day,
            timezone=timezone,
        )
    except CrontabSchedule.MultipleObjectsReturned:
        schedule = CrontabSchedule.objects.filter(
            minute=mailing_date.time.minute,
            hour=mailing_date.time.hour,
            day_of_week=mailing_date.day,
            timezone=timezone,
        ).first()

    task = PeriodicTask(
        name=uuid.uuid4(),
        crontab=schedule,
        task='apps.push.tasks.send_push',
        kwargs=json.dumps({
            'push_pk': self.pk,
            'timezone': timestamp
        })
    )

    return task

def generate_clocked_task(self, mailing_date, timezone, timestamp):
    dt_with_tz = (
        pytz
        .timezone(timezone)
        .localize(datetime.combine(mailing_date.date, mailing_date.time))
    )
    clocked, _ = ClockedSchedule.objects.get_or_create(
        clocked_time=dt_with_tz,
    )
    task = PeriodicTask(
        name=uuid.uuid4(),
        clocked=clocked,
        task='apps.push.tasks.send_push',
        one_off=True,
        kwargs=json.dumps({
            'push_pk': self.pk,
            'timezone': timestamp
        })
```

```
            )

            return task

        @staticmethod
        def mark_tasks_as_general_push(tasks):
            bulk_list = []
            for task in tasks:
                bulk_list.append(GeneralPush(periodic_task=task))
            GeneralPush.objects.bulk_create(bulk_list)

        def delete_mailings_schedule(self):
            self.mailing_schedule.all().delete()
            self.mailing_date.all().delete()

        def create_push_tasks(self, mailing_type):
            # if mailing_type not in (self.MAILING_DAY, self.MAILING_DATE):
            #     return

            tasks = []

            mailing_dates = (self.mailing_schedule.all() if mailing_type == self.MAILING_DAY
                             else self.mailing_date.all())
            all_geo = self.audiences.all().values_list('geo__code', flat=True).distinct()

            for mailing_date in mailing_dates:
                timezones = uniqe_tz.get_unique_timezones(all_geo)

                for timestamp, timezone in timezones.items():
                    if mailing_type == self.MAILING_DAY:
                        task = self.generate_crontab_task(mailing_date, timezone, timestamp)
                    elif mailing_type == self.MAILING_DATE:
                        task = self.generate_clocked_task(mailing_date, timezone, timestamp)
                    else:
                        continue
                    tasks.append(task)

            with transaction.atomic():
                # Delete old tasks
                self.push_tasks.all().delete()

                if tasks:
                    created_tasks = PeriodicTask.objects.bulk_create(tasks)
                    self.push_tasks.set(created_tasks)

            if self.is_general:
                self.mark_tasks_as_general_push(self.push_tasks.all())

            self.save()

        def run_pushing(self, test_mode=False):
            from apps.push.tasks import send_push
            tasks = self.push_tasks.all()
            tasks_count = tasks.count()

            for count, task in enumerate(tasks, start=1):
                print(f'\nTask # {count}/{tasks_count}')
                kwargs = json.loads(task.kwargs)
                kwargs['test_mode'] = test_mode
                send_push(**kwargs)


    class OneSignalNotification(models.Model):
        push = models.ForeignKey(Push, on_delete=models.CASCADE, related_name='os_notifications', db_index=True)
        app = models.ForeignKey(MobileApp, on_delete=models.CASCADE, related_name='os_notifications')
        author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='os_notifications', null=True, db_index=True)
        geo = models.CharField(max_length=4, null=True)
        notification_id = models.CharField(max_length=225, db_index=True)
        external_id = models.CharField(max_length=225, null=True, blank=True)
```

```
    completed_at = models.DateTimeField(auto_now_add=True, db_index=True)
    updated_at = models.DateTimeField(null=True)
    include_players = models.IntegerField(null=True)
    successful_sent = models.IntegerField(null=True)
    converted = models.IntegerField(null=True)
    ctr = models.IntegerField(null=True)

    def __str__(self):
        return f'{self.push} - {self.notification_id}'

    def _calculate_ctr(self):
        try:
            ctr = round(self.converted / self.successful_sent * 100, 2)
            return ctr
        except ZeroDivisionError:
            return 0

    def get_sent_stat(self):
        failed = self.include_players - self.successful_sent
        return [self.include_players, self.successful_sent, failed]

    def update_data(self, data: dict = None):

        if data is None:
            data = onesignal.get_notification(
                notification_id=self.notification_id,
                app_id=self.app.onesignal_id,
                app_api_key=self.app.api_key
            )

        if data:
            try:
                self.include_players = len(set(data.get('include_player_ids', [])))
                self.successful_sent = data.get('successful', 0)
                self.converted = data.get('converted', 0)
                self.ctr = self._calculate_ctr()
                self.updated_at = datetime.now(timezone.utc)

                return self

            except KeyError as e:
                print(f'KeyError - {str(e)}')
                return {'error': f'Notif ID - {self.id}, KeyError - {str(e)}'}

            except AttributeError as e:
                print(f'AttributeError - {str(e)}')
                return {'error': f'Notif ID - {self.id}, AttributeError - {str(e)}'}
class SendTestPushView(View):
    def get(self, request):
        if request.GET.get('success'):
            return render(request, 'send-test-push/success.html')

        apps = MobileApp.objects.values_list('id', 'name')
        languages = Language.objects.values_list('code', 'name')

        return render(
            request,
            'send-test-push/send-test-push.html',
            {
                'apps': apps,
                'push_languages': languages
            }
        )

    def post(self, request):
        data = request.POST.copy()

        app_id = data.get('app')
        user_ip = data.get('user_ip')
```

```
header = data.get('header')
message = data.get('message')
image_url = data.get('image_url')
icon_url = data.get('icon_url')
language = data.get('language')

app = MobileApp.objects.get(pk=app_id)

# Get external_user_id from KT
try:
    kt = WildWildKtDash()
    external_user_id = kt.get_external_id_by_ip(user_ip=user_ip, app_bundle=app.bundle_name)
except IndexError:
    messages.error(request, _('Bad IP address'))
    return JsonResponse({'success': False, 'error': 'Bad IP address'}, status=400)

# Get user data from OS
user_data = onesignal.get_user_data_by_external_id(
    app_id=app.onesignal_id,
    app_api_key=app.api_key,
    external_id=external_user_id
)

# Send test push
translator = Translator()
header = translator.translate(header, language)
message = translator.translate(message, language)

onesignal.test_push(
    app, user_data.player_id,
    image=image_url,
    icon=icon_url,
    header=header,
    message=message
)

return JsonResponse({'success': True})
```