НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних систем, мереж та кібербезпеки

Касаткін Д.Ю., к. пед.н., доц.

Підпис ПІБ, вчене звання і ступінь

___2025 р.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

На тему: Розробка боту для інформування про івенти для молоді у місті з використанням Python

Спеціальність F7 «Комп'ютерна інженерія»

Гарант освітньої програми

к.фіз.-мат.н., доцент _____ Євгеній НІКІТЕНКО

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник випускної бакалаврської роботи

старший викладач _____ Володимир МАТІЄВСЬКИЙ

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав

(підпис) (ПІБ студента)

Київ – 2025

Владислав РОМАШКО

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАНН УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки

Касаткін Д.Ю., к.пед.н., доц. /
підпис ПІБ, вчене звання і ступінь

р.

З А В Д А Н Н Я
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ БАКАЛАВРСЬКОЇ СТУДЕНТА

Ромашко Владислав Олександрович
(прізвище, ім'я, по батькові)

Спеціальність (напрям підготовки): F7 «Комп'ютерна інженерія»

Тема випускної бакалаврської роботи: Розробка боту для інформування про івенти для молоді у місті з використанням Python

Керівник проекту (роботи) Матієвський В.В., старший викладач
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджено наказом ректора НУБіП України від «16» грудня 2024 р. № 2250 «С»

Термін подання завершеної роботи на кафедру 2025, травень, 28
(рік, місяць, число)

Вихідні дані до випускної бакалаврської роботи (дипломного проекту бакалавра)
реалізація пошуку подій з урахуванням країни, міста та типу події

Перелік питань, які потрібно розробити:
Аналіз аналогів, вибір технологій, інтеграція з Ticketmaster API, реалізація Telegram-бота з пошуком подій, використання Aiogram та FSM, тестування функціональності.

Перелік графічних документів (за потреби)

Дата видачі завдання "16" грудня 2024 р.

Керівник бакалаврської роботи _____ Матієвський В.В. старший викладач

Завдання прийняв до виконання _____ Ромашко В. О.
(підпис) (прізвище та ініціали студента)

(підпис) (прізвище та ініціали)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
	Аналіз існуючих рішень для Telegram-ботів	р.	Виконано
	Огляд засобів інтеграції та вибір технологій		Виконано
	Проєктування архітектури та розробка алгоритму роботи бота		Виконано
	Реалізація основного функціоналу Telegram-бота		Виконано
	Тестування та налагодження роботи бота, а також аналіз продуктивності та оцінка ефективності роботи		Виконано

Керівник проекту (роботи) _____ /Володимир МАТІЄВСЬКИЙ/

(підпис) (ініціали та прізвище)

Студент

_____

_____ /Владислав РОМАШКО/

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 56 сторінок, 13 рисунків, 1 таблиця, 7 додатків, 10 джерел.

Об'єктом даного дослідження є створення Telegram-бота, який інформує користувачів про майбутні концерти у їхніх містах.

Предметом дослідження є технології та методи розробки Telegram-ботів із використанням фреймворку Aiogram, а також інтеграція зовнішніх API для отримання актуальної інформації про культурні події.

Актуальність роботи зумовлена тим, що в сучасному світі інформація розповсюджується дуже швидко, але водночас користувачі часто не мають часу самостійно відстежувати цікаві заходи. Telegram-боти стали зручним каналом для отримання своєчасних сповіщень, що робить їх важливим інструментом комунікації. Враховуючи популярність Telegram та потребу у швидкому доступі до інформації про концерти, створення такого бота є актуальним і практично корисним завданням.

Метою роботи є розробка функціонального Telegram-бота з інтуїтивним інтерфейсом, який дозволяє користувачам отримувати оперативну інформацію про концерти у вибраному місті, а також забезпечує стабільну і швидку роботу завдяки застосуванню асинхронного програмування.

У роботі детально розглянуто архітектуру бота, особливості роботи з API Ticketmaster, реалізацію бази даних для збереження інформації про користувачів та їхні запити, а також проведено тестування та оцінку ефективності системи. Практична значущість полягає у створенні зручного інструменту, який допоможе зекономити час користувачів і підвищить їхню обізнаність про культурні події у своїх містах.

Отримані результати можуть слугувати основою для подальшого розвитку подібних систем, зокрема для розширення функціоналу та адаптації до інших типів подій.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у забезпеченні швидкого та зручного доступу до різноманітних даних, як для професійних потреб, так і для особистих інтересів користувачів. У цьому контексті чат-боти набувають все більшої популярності як ефективний засіб комунікації між сервісами та користувачами, значно прискорюючи процеси взаємодії в різних інформаційних сферах. Платформа Telegram, завдяки своїй функціональності та можливості розробки власних сервісів, стала сучасним вибором для багатьох користувачів, які прагнуть не лише до обміну повідомленнями, а й до інтеграції корисних функціональних рішень. Користувачі можуть додавати ботів до груп, каналів або використовувати їх в окремих чатах.

Актуальність роботи зумовлена зростаючим попитом на автоматизовані сервіси, які спрощують пошук та доступ до інформації про події. В умовах швидкого розповсюдження інформації, користувачі часто не мають часу для самостійного відстеження цікавих заходів. Telegram-боти, як зручний канал для отримання своєчасних сповіщень, стали важливим інструментом комунікації [4]. Враховуючи популярність Telegram та потребу у швидкому доступі до інформації про концерти, створення такого бота є актуальним та практично корисним завданням.

Об'єктом дослідження є створення Telegram-бота, який інформує користувачів про майбутні концерти у їхніх містах.

Предметом дослідження є технології та методи розробки Telegram-ботів із використанням фреймворку Aiogram, а також інтеграція зовнішніх API для отримання актуальної інформації про культурні події.

Метою роботи є розробка функціонального Telegram-бота з інтуїтивно зрозумілим інтерфейсом, який дозволяє користувачам отримувати оперативну інформацію про концерти у вибраному місті, а також забезпечує стабільну і швидку роботу завдяки застосуванню асинхронного програмування.

Для досягнення поставленої мети були визначені наступні завдання:

Провести аналіз існуючих рішень на ринку щодо інформування про івенти.

Обґрунтувати вибір технологій та програмних інтерфейсів (API), що будуть використані для розробки бота.

Спроєктувати архітектуру Telegram-бота та розробити алгоритм його роботи.

Реалізувати основний функціонал Telegram-бота, включаючи пошук подій з урахуванням країни, міста та типу події.

Виконати тестування та налагодження роботи бота, а також аналіз продуктивності та оцінка ефективності роботи.

Бакалаврська кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

Розділ 1 присвячений аналізу існуючих рішень для інформування про івенти, обґрунтуванню вибору мови програмування Python та бібліотеки Aiogram, а також огляду інших використаних технологій та API.

Розділ 2 охоплює архітектуру Telegram-бота, опис алгоритму його роботи та деталізацію взаємодії з базами даних та API.

Розділ 3 містить інформацію про тестування функціоналу бота, оцінку ефективності роботи бота та визначення напрямів подальшого вдосконалення.

Висновки підсумовують результати виконаної роботи, а список використаних джерел та додатки містять посилання на використані матеріали та вихідний код програми

Наукова новизна роботи полягає у систематизації та адаптації сучасних підходів до розробки асинхронних Telegram-ботів із застосуванням фреймворку Aiogram для вирішення задачі оперативного інформування про локальні події. Зокрема, розроблена система демонструє ефективну інтеграцію з зовнішніми джерелами даних (Ticketmaster API) та реалізацію механізму FSM для побудови складних діалогових сценаріїв, що підвищує зручність користувача порівняно з існуючими аналогами, які часто мають обмежені можливості фільтрації та географічного охоплення.

Практична значущість роботи полягає у створенні зручного інструменту, який допоможе зекономити час користувачів і підвищить їхню обізнаність про культурні події у своїх містах. Розроблений бот автоматизує рутинні процеси та сприяє подальшому зростанню цифрових сервісів. Отримані результати можуть слугувати основою для подальшого розвитку подібних систем, зокрема для розширення функціоналу та адаптації до інших типів подій. Впровадження Telegram-бота для знаходження різних типів подій продемонструє можливості інтеграції сучасних технологій інформаційних сервісів і дозволить створити корисний інструмент, який буде корисним для широкого кола користувачів.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ПРОГРАМНИХ ЗАСОБІВ

Аналіз ринку існуючих рішень, щодо інформування про івенти

Перед безпосередньою розробкою Telegram-бота для інформування про івенти для молоді у місті, критично важливим є всебічний аналіз поточного ринку існуючих рішень. Цей етап дозволяє ідентифікувати успішні підходи, виявити прогалини в функціоналі та визначити потенційні напрямки для інновацій. Дослідження існуючих аналогів дасть змогу сформувати вимоги до майбутнього програмного продукту, забезпечуючи його конкурентоспроможність та відповідність потребам цільової аудиторії.

На ринку інформування про події представлено значну кількість сервісів та платформ, що використовують різні канали комунікації, включно з веб-сайтами, мобільними додатками та чат-ботами.

Веб-платформи та агрегатори івентів: До таких належать глобальні гіганти, як Eventbrite, Meetup, та українські аналоги, наприклад, Concert.ua та Ці платформи пропонують широкий спектр подій, від концертів та виставок до освітніх заходів та тренінгів. Їхні переваги включають велику базу даних подій, систему фільтрації за категоріями, датами та локаціями, а також можливість купівлі квитків. Однак, їхній недолік полягає у необхідності активної дії з боку користувача – відвідування веб-сайту для пошуку інформації.

Мобільні додатки для подій: Багато з вищезгаданих веб-платформ мають власні мобільні додатки, що забезпечують зручніший доступ до інформації та сповіщення про події. Прикладами можуть бути додатки від Bandsintown (орієнтований на музичні події), Dice або локальні додатки, що концентруються на подіях у конкретних містах. Перевагами є push-сповіщення та персоналізовані рекомендації, але вони вимагають встановлення окремого додатку, що може бути

незручним для користувачів, які прагнуть мінімізувати кількість встановлених програм.

Згідно з процесом аналізу, більшість існуючих ботів ефективно використовують API сторонніх сервісів (наприклад, Eventbrite API, Google Calendar API або спеціалізовані API для музичних подій) для збору та агрегації інформації про події.

Основні переваги:

Швидкість надання інформації: Чат-боти дозволяють користувачам миттєво отримувати необхідну інформацію без зайвих кліків чи переходів.

Простота використання: Інтерфейс чат-бота інтуїтивно зрозумілий, оскільки базується на діалоговій моделі, що є звичною для більшості користувачів месенджерів.

Можливість інтеграції: Багато ботів інтегруються з іншими популярними сервісами (наприклад, Google Maps для навігації, сервіси продажу квитків), що розширює їхній функціонал.

Персоналізація (часткова): Деякі боти пропонують базові можливості персоналізації, такі як вибір міста або категорій подій.

Основні недоліки:

Обмежений вибір міст та локацій: Значна частина ботів орієнтована на великі міста або має обмежену географічну базу даних, що є критичним недоліком для мешканців менших населених пунктів або для тих, хто шукає події у конкретних районах міста.

Відсутність гнучких налаштувань та фільтрації: Багато існуючих рішень не надають достатньої гнучкості у налаштуванні фільтрів (наприклад, за віком, інтересами, часом доби, ціновою категорією), що призводить до отримання надмірної або нерелевантної інформації.

Залежність від зовнішніх API: Якість та повнота інформації бота безпосередньо залежить від доступності та актуальності даних, що надаються зовнішніми API. У разі зміни політики або припинення роботи одного з таких API, функціонал бота може бути порушений.

Відсутність агрегації з локальних джерел: Більшість ботів ігнорують локальні джерела інформації (наприклад, оголошення в місцевих спільнотах, афіші невеликих закладів), що може призвести до втрати значної кількості релевантних для молоді подій.

Чат-боти в месенджерах саме цей сегмент ринку є найбільш релевантним для даної бакалаврської роботи. Чат-боти в месенджерах, таких як Telegram, WhatsApp, Viber, набувають популярності завдяки своїй простоті використання та доступності. Серед відомих рішень у Telegram можна виділити:

Bandsintown Bot: Цей бот є інтеграцією однойменного сервісу і дозволяє користувачам отримувати інформацію про концерти у вибраних містах та відстежувати улюблених виконавців. Його переваги – велика база даних музичних подій та зручність використання. Недоліки можуть включати обмеження лише музичними подіями та відсутність гнучких налаштувань для інших типів івентів.

ConcertBuddy: Бот, орієнтований на пошук концертів та музичних подій. Він часто пропонує функціонал сповіщень та інтеграцію з сервісами продажу квитків.

EventBot (різні імплементації): Існує безліч ботів з назвою "EventBot", розроблених для різних цілей – від інформування про корпоративні заходи до публічних подій. Їхній функціонал може сильно відрізнятися, але загальною перевагою є можливість швидкого отримання інформації без необхідності переходу на зовнішні ресурси.

Перед тим як почати створення Telegram-бота, який буде інформувати користувачів про концерти та події, варто уважно вивчити існуючі аналоги на ринку. Серед відомих рішень можна виділити таких ботів, як Bandsintown, ConcertBuddy та EventBot. Вони ефективно використовують платформу Telegram для розповсюдження інформації.

У цій роботі буде зроблено акцент на виявлених перевагах і недоліках, зосереджуючись на створенні найзручнішого та простого в користуванні

Telegram-бота, який підтримуватиме широкий вибір міст і надаватиме можливість персоналізованих налаштувань.

.2. Вибір технологій та API

Мета полягає у тому, щоб створити Telegram-бота, який буде корисний тим, що інформує користувачів про концерти, виступи, змагання та інші події у вибраному ним країні та місті. Для цього я використовував підходящі технології. Вони чітко підходять під вимоги мого функціоналу для бота та задовольняють потреби для стабільної, гнучкої та продуктивної роботи системи.

Обравши мову програмування Python, ми впевнилися, що це найкращий вибір і зараз коротко наведемо аргументи чому [3], [6]. За допомогою цієї мови програмування, можна без всяких проблем інтегрувати API, а також бази даних, які мають спеціалізовані бібліотеки Telegram API. Саме для коду це є проста та підтримувана база, через те, що вона має багато готових рішень та зрозумілу інструкцію.

Для зовнішніх API для отримання інформації про концерти використовується один із найвідоміших сервісів для отримання даних про заходи та продаж квитків – Ticketmaster API [1], [5]. Я також хотів використати API , який містить базу даних про музичні події та дозволяє шукати заходи за виконавцем чи містом, але у ході вивчення даного API – вони перестали надавати такий доступ. Це був не дуже приємний досвід у пошуку потрібних API, адже інших варіантів мені не вдалося знайти. Тому, єдиний вихід із мої ситуації був

Також, не менш важливим є те, що слід розглянути надсилання запитів HTTP і обробка відповідей у форматі JSON, яка дозволяє інтегруватися з цим сервісом.

Бот Telegram використовує API – для взаємодії з користувачами [2]. Такі бібліотеки, як Telebot і Aiogram, підтримують його. Великою перевагою для

ефективне керування потоками даних і високу продуктивність. А Telebot простішій у використанні та підходить для розробників невеликих ботів з базовими функціями.

На мою думку, обійти стороною функцію для збереження налаштувань користувачів та їхніх запитів – є безглуздо, тому я використовував SQLite. Це проста база даних, яка не вимагає окремого серверного середовища, ідеально підходить для Telegram-ботів, які працюють у локальному режимі або на невеликих хостингових ресурсах.

Як вказували раніше всі ці методи, функції та сервіси створюють зручного і ефективного бота, що дає погляд на його сучасну автоматизовану систему інформування.

Огляд засобу інтеграції telegram api – aiogram

Уявіть, що ви запускаєте бота, а він ніби потай працює за кулісами, миттєво відгукується на кожного користувача і без проблем справляється з десятками одночасних завдань — це як вулик, в якому всі бджілки (запити) безперешкодно літають і збирають мед (відповіді), не забиваючи запуск. Саме так працює

По суті, Aiogram — це ваш асинхронний помічник у світі Telegram-ботів. У той час як простий бот чекає, поки код попрацює з одним користувачем, Aiogram одночасно відкриває з ним десяток «вікон» і в кожному слухає, що відбувається: команду, натискання кнопки, нове повідомлення. Він не відволікається і не «гальмує», бо в основі лежить asyncio — як черга на касі з багатьма терміналами, де кожен клієнт швидко обслуговується, а черга рухається без зупинок.

Коли ви ставитеся до кожного запиту як до окремої маленької задачі, вам не доводиться переписувати тони коду: Aiogram уже пропонує готові хендлери для команд, можливість будувати маршрутизацію через callback-кнопки і навіть влаштовувати багатокрокові діалоги з користувачем завдяки станам (FSM). Ніби

ви склали набір конструкторів: берете потрібні блоки — і ваш бот за мить «оживає».

А ще у Aiogram є вбудовані фільтри, аби легко відсівати непотрібні повідомлення, та інтеграція з aiohttp, щоб за лічені рядки налаштувати звернення до будь-яких зовнішніх API. Логування, обробка помилок, тонке налаштування часу очікування — усе вже готове “з коробки”.

Завдяки цьому Aiogram стає не просто фреймворком, а «довіреним соратником» розробника: поки він бере на себе технічні дрібниці, ви можете зосередитися на найцікавішому — придумувати і втілювати у життя власні ідеї для бота.

Також, слід додати, що спільнота навколо Aiogram відчувається буквально за кроком: щойно виникне питання, я відкриваю чат у Telegram, запитую — і вже за кілька хвилин хтось із ветеранів фреймворка підкаже, як обійти помилку або реалізувати нетипову фічу. Пам’ятаю, як уперше зіткнувся з обробкою inline-кнопок — досі гостро відчував невпевненість, а в офіційному чаті Aiogram за кілька годин пояснили все до дрібниць і навіть скинули робочий приклад.

На GitHub-форджі Aiogram теж завжди кипить життя: кожен день з’являються свіжі пул-реквести — хтось адаптує код під новий метод Bot API, хтось виправляє дрібні баги або додає корисні утиліти. Я сам неодноразово долучався: із запитом про оптимізацію запитів із Ticketmaster — і вже за кілька днів бачив власний патч у новій версії.

Коли Telegram раптово випустив підтримку голосових чатів у групах, учасники фреймворка буквально за тиждень інтегрували це в Aiogram, і я не встиг здивуватися, як мій бот вже вмів реагувати на ці нові події [9]. Анонси всіх змін і короткі гіди виходять у офіційному каналі оновлень: там не лише сухі нотатки, а й живі приклади коду, коментарі від авторів та поради щодо оптимального використання.

Саме ця атмосфера — коли замість довгого очікування відповіді ти отримуєш готове рішення «з перших рук» — робить роботу з Aiogram такою

приємною. Не просто фреймворк, а дружня спільнота однодумців, яка швидко реагує на зміни Telegram API і допомагає тобі крокувати в ногу з технологіями.

Основні концепції фреймворку

Коли ви починаєте знайомитися з Aiogram, вас може вразити, наскільки чітко побудована його внутрішня логіка: відразу відчувається, що автори фреймворку дбали про зручність розробника, неначе проект створювався для себе. Ось кілька ключових ідей, які лежать в основі цього рішення:

er як координатор подій

Dispatcher — це серце Aiogram, його «контрольний центр». Уявіть, що ваш бот — це невеликий офіс, а Dispatcher — секретарка, яка приймає всі вхідні дзвінки (апдейти), розпізнає, хто саме телефонує, і скеровує звернення до відповідного співробітника (хендлера). Саме завдяки Dispatcher ви можете ефективно організувати маршрутизацію команд, callback-кнопок чи просто текстових повідомлень, не турбуючись про «гарячу лінію».

2. Хендлери як «фахівці» за профілем

Кожен хендлер (handler) — це окрема функція, що виконує одну чітко визначену задачу: обробити команду /start, відповісти на натискання кнопки або зреагувати на певний текст. Коли до бота приходить повідомлення, Dispatcher передає його саме тому хендлеру, який «спеціалізується» на такому запиті. Завдяки цьому ваш код залишається чистим і зрозумілим — як у справжньому колективі, де кожному працівнику відведена своя ділянка роботи.

3. Фільтри для відсіювання шуму

Aiogram пропонує набір фільтрів (filters), які допомагають відокремити корисні запити від стороннього «шуму». Наприклад, ви швидко вказуєте: «Приймати тільки фото», «реагувати тільки на текстові повідомлення довжиною більше 10 символів» або «відповідати лише на ті callback-запити, що починаються з event:». Це схоже на вахтера у великому будинку — він запускає лише тих гостей, які мають запрошення.

4. Middleware — проміжні “приклади” коду

Якщо хендлери — це кінцеві працівники, то Middleware — це помічники, які проходять перед Dispatcher: реєструють факт звернення, перевіряють авторизацію, можуть змінити запит або додати контекст. Для мене це як охоронець біля входу, який перевіряє документи, перед тим як передати людину далі.

5. FSM (Finite State Machine) для багатокрокових діалогів

Дуже зручна концепція — кінцевий автомат станів. Уявіть спорядження туриста: він проходить пункт «країна» → «місто» → «тип події». Кожен крок — це чітко визначений стан, і поки користувач не вибрав «місто», бот не перейде до наступного етапу. FSM дозволяє будувати складні, але контрольовані сценарії, де користувач неначе слідує покроковій інструкції [7], [10].

6. Інтеграція з asyncio і aiohttp

Насправді головна родзинка Aiogram — це не просто чередувач хендлерів, а глибока інтеграція з асинхронним стеком Python [8]. Усе, починаючи від обробки повідомлень і закінчуючи запитами до зовнішнього API через aiohttp, працює в одному великому циклі подій. Розробник пише звичні await виклики, а під капотом запускаються тисячі корутин, кожна з яких обробляє свій шматок роботи.

Разом ці концепції перетворюють Aiogram із звичайного набору інструментів на справжню екосистему для створення ботів. Коли я працюю над складними сценаріями — чи то багатокроковий пошук подій, чи зв’язок із кількома API — мені не доводиться вигадувати «костилі»: все необхідне вже є «з коробки», а фреймворк наче дружній колега, який завжди підкаже найкращий шлях.

Асинхронний підхід: основа роботи Aiogram

Асинхронність у Aiogram організована навколо event loop з використанням бібліотеки asyncio. Кожен хендлер у цьому фреймворку запускається як корутина — «легка» функція, яка автоматично призупиняється під час очікування відповіді від Telegram API чи зовнішнього сервісу і відновлює роботу одразу після

отримання даних. Під час таких пауз бот продовжує обробляти нові повідомлення, не допускаючи ніяких затримок.

Мій досвід показує: саме підхід із корутинами дозволяє Aiogram залишатися чуйним навіть у моменти пікового навантаження. Якщо кілька користувачів одночасно відправляють команди, кожен запит обробляється незалежно, тож повільний виклик до стороннього API ніяк не позначається на швидкості реакції бот-системи в цілому.

До ключових переваг такого рішення належать:

птимальне використання ресурсів. Корутини створюються швидко та майже не споживають пам’яті, тому бот може одночасно обслуговувати велику кількість підключень.

нучкість налаштувань. Параметри таймаутів і обсяг очікування легко коригувати під вимоги конкретного API чи потужності сервера.

розумілий код. Виклики await роблять логіку хендлерів лінійною та чистою, без складної організації потоків.

Отже, асинхронна модель у Aiogram — це не просто технічна особливість, а справжній фундамент, що гарантує стабільність і масштабованість бота. Під час розробки власного проєкту я переконався: можливість відокремлювати час очікування зовнішніх запитів від обробки нових звернень робить систему більш надійною та легкою в підтримці.

1.4 Огляд бібліотек використаних при створенні боту

На початкових етапах розробки Telegram-бота для інформування про події в місті, здавалося, що завдання полягатиме виключно у зв'язуванні декількох програмних інтерфейсів (API) та імплементації логіки команд [6]. Однак, згодом стало очевидним, що вибір відповідних інструментів є набагато важливішим аспектом, оскільки він визначає підтримуваність, масштабованість та подальший розвиток програмного коду. У цьому розділі буде представлено досвід роботи з

ключовими бібліотеками, використаними в проєкті, з детальним обґрунтуванням їхнього вибору, а також пояснено причини відмови від альтернативних рішень, попри їхню популярність.

Почну з головного героя цього проєкту — бібліотеки Aiogram. Вона стала основою всього застосунку. Aiogram — це сучасний Python-фреймворк, створений спеціально для роботи з Telegram Bot API. Його головна особливість — асинхронність. Завдяки цьому бот може обробляти кілька запитів одночасно без зависань і блокувань.

Ми використовуємо Aiogram не тільки для обробки повідомлень і команд, а й для побудови повноцінного сценарію взаємодії з користувачем. Наприклад, через FSM (Finite State Machine, тобто “машину станів”) мені вдалося реалізувати покроковий пошук подій: спочатку країна, потім тип події, потім підтвердження. І все це — структуровано, збережено в контексті, без плутанини.

Що конкретно було використане:

- FSMContext — для керування контекстом користувача на кожному етапі.
- Message та CallbackQuery — щоб ловити повідомлення й відповіді на кнопки.
- CommandStart, Command — фільтри для розпізнавання стандартних команд.

  patcher, Bot — основні об’єкти, через які бот «слухає» і «говорить».

Чому це було зручно? Усе логічно розбито по файлах: обробники команд, клавіатури, утиліти, API. У результаті — не просто бот, а реальний проєкт із модульною архітектурою, який легко зрозуміти й підтримувати. І ще один величезний плюс: спільнота Aiogram дуже активна. Якщо щось не працює, майже завжди можна знайти відповідь на GitHub або в Telegram-чатах див. рис.

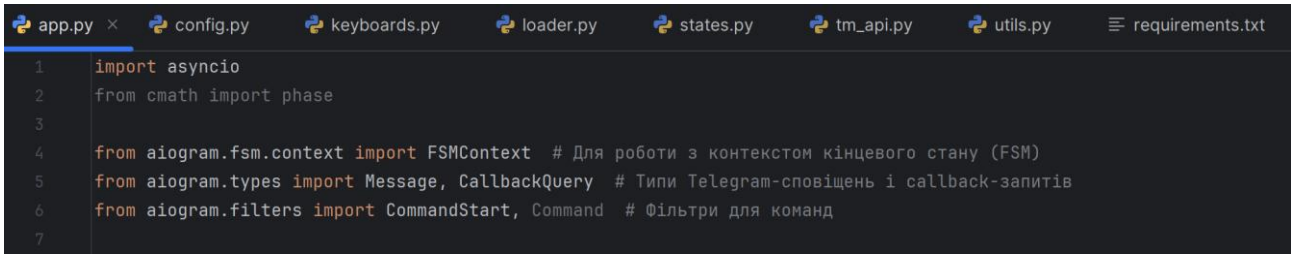

Рисунок 1. Бібліотеки Aiogram

Оскільки Aiogram базується на асинхронному виконанні, без asyncio тут ніяк. Це стандартна бібліотека Python, яка відповідає за створення та керування асинхронними задачами (coroutines). Завдяки їй мій бот може, наприклад, одночасно:

- обробляти команди від кількох користувачів;
- звертатися до API Ticketmaster;
- відповідати або редагувати повідомлення з фото.

Важливо, що при цьому нічого не блокується, і бот не "випадає з реальності", як це буває в синхронних сценаріях.

Модулі власної розробки: keyboards, loader, states, utils

Коли логіка бота почала розростатися, стало зрозуміло, що все тримати в одному файлі — нереально. Тому я розбив функціонал на окремі модулі:

keyboards.py — тут описані всі клавіатури, які бачить користувач: вибір країни, типу події, перехід назад тощо. Використовуються inline-кнопки, які надсилають callback'и. представлено на рис. 2

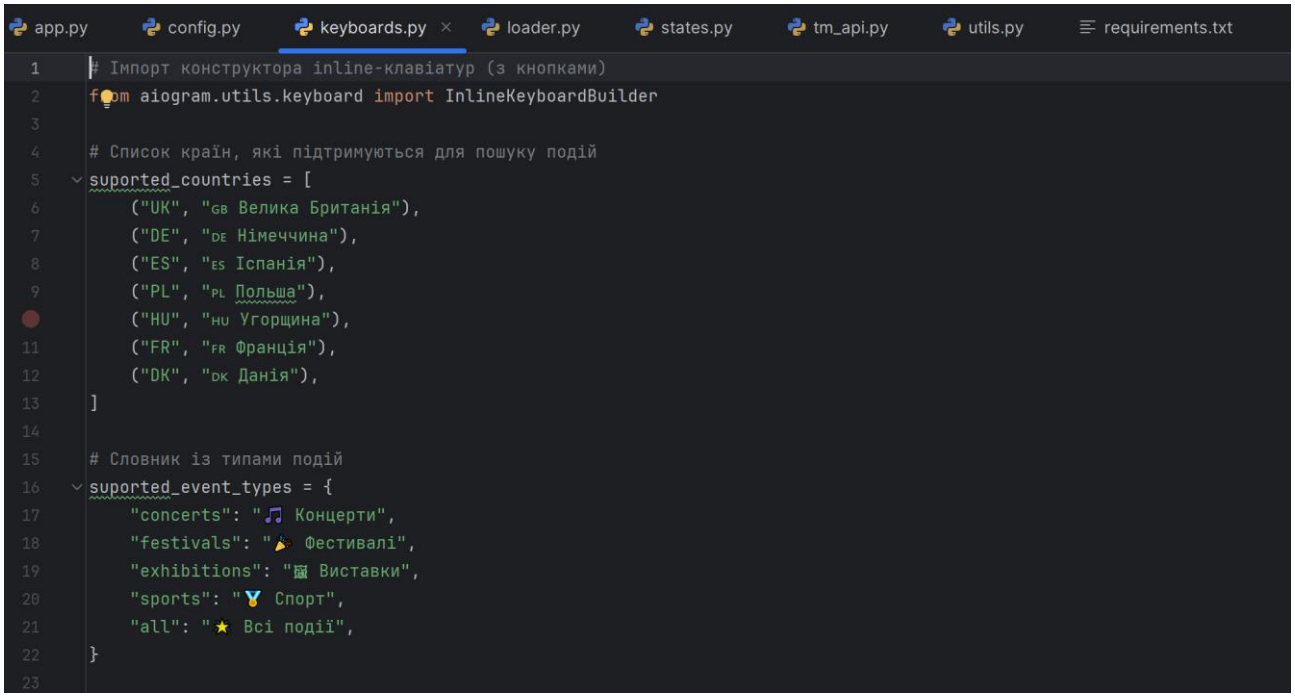

Рисунок 2. Коротка демонстрація коду з файлу keybords.py

loader.py — це як центральний вузол. Тут ініціалізується bot, dispatcher (dp) і об'єкт для взаємодії з Ticketmaster API дивись на рис. 3

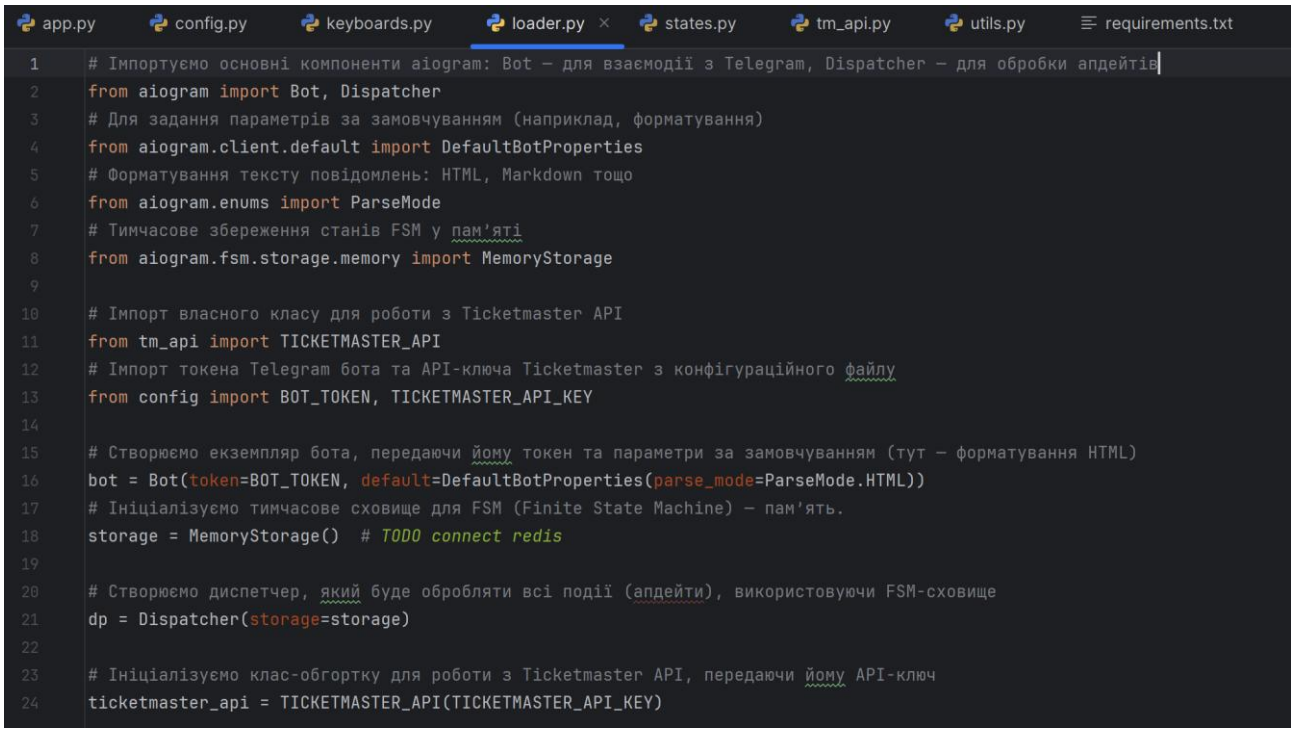

Рисунок 3. Демонстрація в програмі коду з файлу loader.py

states.py — оголошення станів машини FSM. Наприклад, SearchState має country, event_type, confirmation — тобто етапи, через які проходить користувач під час пошуку подій.

app.py | config.py | keyboards.py | loader.py | states.py | tm_api.py | utils.py | requirements.txt

```python
# Імпортуємо базові класи для роботи з машиною станів (FSM) з бібліотеки aiogram
from aiogram.fsm.state import StatesGroup, State


# Визначаємо групу станів для сценарію пошуку подій
class SearchState(StatesGroup):  7 usages
    # Стан, у якому користувач обирає країну (наприклад, "UK", "DE" тощо)
    country = State()

    # Стан, у якому користувач вводить або обирає місто
    city = State()

    # Стан, у якому користувач обирає тип події (наприклад, концерт, фестиваль тощо)
    event_type = State()

```

Рисунок 4. Демонстрація в програмі коду з файлу states.py

utils.py — утилітний файл. Там знаходиться, зокрема, функція answer_or_edit_with_or_without_photo, яка розумно вирішує: відповідати новим повідомленням чи редагувати попереднє. Це дуже зручно, бо Telegram іноді обмежує кількість змін повідомлення.

app.py | config.py | keyboards.py | loader.py | states.py | tm_api.py | utils.py | requirements.txt

```python
# Імпортуємо необхідні типи з aiogram для роботи з повідомленнями
from aiogram import types


# Асинхронна універсальна функція для відповіді або редагування повідомлення, з фото або без
async def answer_or_edit_with_or_without_photo(  4 usages
        message: types.Message | types.CallbackQuery, # Повідомлення або callback-запит
        photo: str | types.InputFile | None,    # Шлях до фото або об'єкт InputFile, або None
        text: str,      # Текст повідомлення
        reply_markup=None):     # Клавіатура (може бути None)

    # Якщо прийшов CallbackQuery, витягуємо оригінальне повідомлення
    if isinstance(message, types.CallbackQuery):
        message = message.message


    # Якщо є фото
    if photo:
        # Якщо вже є повідомлення з фото, і воно надіслане ботом — редагуємо лише медіа (фото і підпис)
        if message.content_type == "photo" and message.from_user.is_bot:
            await message.edit_media(types.InputMediaPhoto(media=photo, caption=text), reply_markup=reply_markup)
        else:
            if message.from_user.is_bot:
                    await message.delete()

            await message.answer_photo(photo, caption=text, reply_markup=reply_markup)

    # Якщо фото немає
    else:
        # Якщо бот надіслав старе повідомлення — видаляємо його
        if message.content_type == "photo" and message.from_user.is_bot:
            if message.from_user.is_bot:
                await message.delete()
            # Відправляємо нове текстове повідомлення
            await message.answer(text, reply_markup=reply_markup)
        else:
            # Якщо повідомлення надіслав бот — редагуємо його текст
            if message.from_user.is_bot:
                await message.edit_text(text, reply_markup=reply_markup)
            else:
                # Якщо повідомлення не від бота — просто надсилаємо нове повідомлення
                await message.answer(text, reply_markup=reply_markup)
```

Рисунок 5. Коротка демонстрація в програмі коду з файлу utils.py

Порівняння бібліотеки Aiogram з іншими аналогами

Під час розробки Telegram-бота розробник може зіштовхнувся з доволі простим, на перший погляд, питанням: яку бібліотеку використовувати? Спочатку навіть не думав, що це буде настільки важливо. Але в процесі роботи стало зрозуміло: вибір фреймворку впливає буквально на все — як на швидкість розробки, так і на якість коду, стабільність бота, гнучкість функціоналу тощо.

Я зупинився на Aiogram, і вже на практиці побачив, що це був дуже вдалий вибір. Для чесності — я все ж порівняв його з іншими популярними рішеннями: python-telegram-bot, Telethon та Pyrogram. У кожної з них є свої плюси, але для мого проєкту (бота, який повідомляє про події в місті через Telegram) Aiogram виявився найбільш логічним і зручним варіантом.

Чому саме він? По-перше, асинхронність. Так, довелося трохи розібратися з asyncio, але в результаті я отримав бота, який працює швидко і не зависає навіть при одночасних запитах. По-друге, FSM — дуже крута річ, коли треба чітко контролювати послідовність дій користувача. По-третє, структура самого коду. Вона гнучка, розбита на модулі, і все логічно розкидано по файлах, що значно полегшує підтримку.

Для порівняння: python-telegram-bot — класна бібліотека для початку, але вже на середині реалізації мого бота я зрозумів, що мені бракує гнучкості [7], . А от Telethon та Pyrogram — це вже зовсім інша історія. Вони працюють не з Bot API, а з повноцінним Telegram-клієнтом, що відкриває багато можливостей, але й ускладнює життя. Мені потрібен був саме бот, а не юзербот чи інструмент для автоматизації.

Що ще сподобалось у Aiogram — це активна спільнота. Коли щось ламалось або не працювало (а таке було, чесно кажучи, не раз), завжди можна було знайти рішення на GitHub або спитати в Telegram-чаті. Це дуже підтримує, коли сидиш вночі і ловиш якусь дурну помилку в логіці.

Підсумовуючи: Aiogram — це не просто бібліотека, це інструмент, який реально допомагає будувати складні речі просто. І що найважливіше — він не заважає. Він гнучкий, легкий і добре підходить для масштабування. Якби я починав усе спочатку — все одно вибрав би Aiogram. Без сумнівів.

Нижче наведена таблиця-порівняння, яка вказує на основні переваги та недоліки різних бібліотек:

Таблиця 1. Порівняння бібліотек для Telegram-ботів

Назва бібліотеки				
Тип	Фреймворк для ботів	Фреймворк для ботів	Telegram API клієнт	Telegram API клієнт
Підхід до роботи	Асинхронний	Синхронний і частково асинхронний (з версії 13+)	Повністю асинхронний	Підтримка як sync, так і
Призначення	Боти (Bot API)	Боти (Bot API)	Повний доступ до Telegram	Повний доступ до Telegram
Простота використання	Сучасний, зручний DSL, але вимагає знань asyncio	Найпростіший для новачків	Складніший, потребує розуміння Telegram-протоколу	Простий для клієнтів Telegram, але не для ботів
Робота з ботами	Основна мета	Основна мета	Частково (через userbot або бот-токен)	Частково (через userbot або бот-токен)
Підтримка FSM (машина станів)	Вбудована FSM	Через сторонні рішення	Немає	Немає
Гнучкість	Висока (завдяки	Середня	Дуже висока (повний доступ до	Висока
Документація	Хороша, активно оновлюється	Відмінна, з прикладами	Складна, менш дружня до новачків	Добра, але більш технічна
Швидкість виконання	Висока (завдяки	Нижча (у sync-режимі)	Висока	Висока
Підтримка	Повна підтримка	Повна підтримка	Часткова	Часткова
Можливість писати userbot'ів	Немає	Немає	Так	Так

Активність проєкту на	Активно розвивається	Дуже активний	Активний, але з меншими оновленнями	Активний
Використання у великих проєктах	Часто використовується	Дуже поширений	Рідше (переважно для automation)	Часто для клієнтів та автоматизації

Висновок до розділу 1

У даному розділі дипломної роботи розглянуті ключові аспекти предметної області, пов’язані зі створенням Telegram-ботів для інформування про культурні події, зокрема концерти. Здійснено ґрунтовний аналіз ринку вже існуючих рішень, серед яких було виділено як масштабні платформи (Eventbrite, Meetup, Concert.ua), так і окремі Telegram-боти (Bandsintown Bot, ConcertBuddy, EventBot). Дослідження дозволило виявити переваги подібних систем — на зразок швидкості доступу до інформації, інтегрованості з месенджером та простоти у використанні. Водночас було ідентифіковано низку обмежень, серед яких — брак персоналізації, обмежене охоплення міст, недостатня деталізація подій та залежність від сторонніх API.

Особливу увагу приділено вибору мов програмування, інструментів і бібліотек, що використовуються для створення функціонального Telegram-бота. У межах технічного обґрунтування було обрано мову Python — як оптимальне рішення з огляду на її гнучкість, читаємість коду та розвинену екосистему бібліотек. На практиці це дало змогу реалізувати як роботу з Telegram Bot API, так і взаємодію з зовнішніми сервісами (у тому числі через HTTP-запити до Ticketmaster API). Для реалізації діалогових сценаріїв, керування станами користувача та обробки запитів у реальному часі використано асинхронний фреймворк Aiogram, який став основою архітектури всього застосунку.

Використання асинхронного підходу (asyncio) забезпечило стабільну роботу бота при одночасному обслуговуванні декількох користувачів. Це рішення дозволяє уникати блокування потоку виконання при зверненні до зовнішніх

сервісів — зокрема, до Ticketmaster API, який не завжди повертає відповідь миттєво. У таких випадках корутина (асинхронна функція) зупиняється і звільняє потоки для обробки інших запитів, що значно покращує продуктивність системи та загальну «відповідну здатність» (responsiveness) користувача.

Особливо важливим для майбутньої реалізації стало використання машини скінченних станів (FSM), яка дозволяє будувати чіткі, послідовні сценарії взаємодії з користувачем. Замість того, щоб обробляти всі повідомлення в одному місці, діалоговий процес розбитий на логічні етапи: вибір країни → місто → тип події. Завдяки FSM знижується ймовірність логічних помилок, а користувач завжди отримує релевантну відповідь на поточному етапі взаємодії.

Крім того, в межах даного розділу було обґрунтовано доцільність використання бази даних SQLite — як локального рішення для збереження користувацьких налаштувань (наприклад, вибране місто). Це дозволило реалізувати персоналізацію без додаткових витрат на зовнішні сервіси або складні SQL-сервери, що особливо актуально в умовах обмежених ресурсів студентського проєкту.

Також порівняно кілька бібліотек для роботи з Telegram API — Aiogram, Telebot, python-telegram-bot, Telethon, Pyrogram. У результаті аналізу технічних можливостей, архітектури та документації саме Aiogram було визнано найбільш придатним для поставлених завдань. Зокрема, через підтримку FSM, можливість легкої інтеграції з asyncio та aiohttp, логічну структуру коду та активну спільноту з великою кількістю прикладів.

Загалом проведений аналіз підтвердив доцільність обраного технічного стеку та дозволив сформувати чітке бачення майбутньої архітектури системи. Отримані висновки стали основою для наступного етапу — проєктування та реалізації Telegram-бота, що буде представлено в розділі 2. Прийняті технічні рішення орієнтовані на ефективність, масштабованість та зручність для кінцевого користувача.

РОЗДІЛ 2 ПРОЄКТУВАННЯ І РЕАЛІЗАЦІЯ БОТА

.1. Архітектура Telegram-бота

На початкових етапах розробки даного бота було відзначено високий рівень модульності та продуманості кожного компонента. Це дозволило ефективно інтегрувати окремі фрагменти коду, забезпечуючи їх взаємодію та формування цілісної функціональної системи. Практична реалізація цього підходу демонструється наступним чином:

pp.py – центр керування

Це серце бота, тут зібрано маршрутизацію команд, колбеків і FSM-логіку. Саме тут запускається сценарій: користувач натискає «Пошук події» обирає країну вводить місто вибирає тип події бачить список і деталізацію. Працюючи з цим файлом, відчуваєш, як логіка «проходить» через послідовні стани й акуратно повертається результат.

oader.py – майстер з налаштувань

У цьому файлі створюються об’єкти Bot, Dispatcher і клієнт Ticketmaster API. Завдяки чіткому розділенню, коли я запускаю бота, всі компоненти вже «приведені до ладу» й готові до роботи. Якщо знадобиться замінити MemoryStorage на Redis — достатньо однієї зміни тут, і весь бот мігрує без зайвого клопоту.

tates.py – провідник користувача

Шлях від вибору країни до типу події побудовано як кінцевий автомат із трьома станами [7]. Я побачив, що завдяки FSM користувач ніколи не «заблукає» в діалозі: бот чітко знає, на якому кроці ми зараз, і від цього неможливо відступити без завершення поточної послідовності.

eyboard.py – голос інтерфейсу

Всі inline-клавіатури знаходяться в одному місці. Мені подобається, що зміни тексту кнопок або їх кількості вносяться за лічені хвилини. Коли я додавав пагінацію, достатньо було скоригувати одну функцію — і весь потік переходів між сторінками запрацював “як годинник”.

m_api.py – міст до зовнішнього світу

Клас для асинхронних запитів до Ticketmaster API. Він незмінно викликає довіру: чистий інтерфейс, чітке формування URL, обробка результатів. Я вже звик бачити тут свіжі дані про концерти за кілька мілісекунд.

tils.py – дрібнички, що роблять велику справу

Утиліта answer_or_edit_with_or_without_photo перетворює «кашу» з фото й тексту на акуратні відповіді або редагування повідомлень. Особисто я завжди тішуся, коли бот не додає зайві меседжі в чат, а просто модифікує вже існуюче — це робить діалог чистим і приємним.

– зберігачі секретів

Тут приховані всі токени й ключі. За допомогою dotenv я можу тримати свої секрети в безпеці, не турбуючись, що випадково викладу їх у публічний репозиторій.

Кожен модуль ніби живе своїм життям, але завдяки чітким «контактним точкам» (хендлерам, методам API, станам) вони працюють узгоджено, як злагоджений оркестр [7], [9]. Працювати з такою архітектурою приємно — можна зосередитися на логіці, не «копаючись» у нескінченних залежностях. Такий

дизайн дає впевненість: якщо завтра захочу додати нову фічу або підключити інший API, знадобиться мінімум зусиль, і код залишиться чистим і зрозумілим.

Текст усіх файлів представлено додатках А-Є.

.2. Опис алгоритму роботи

Починається все, як і в більшості ботів — із команди /start. Але навіть тут я замислився: як написати вітання, щоб воно не звучало як сухий текст? Щоб це було ненав'язливо, але зрозуміло. Я хотів, щоб користувач відчув: «Окей, цей бот зроблений не абияк, він продуманий». І вже з першого повідомлення бот пропонує меню з вибором дій: шукати події, змінити місто, або ознайомитись із автором. Простий набір опцій, але вони — як двері до основної логіки.

Далі все зав'язано на основному сценарії — пошуку подій. Тут починається найцікавіше. Користувач вводить місто або обирає з меню, а бот у цей момент виконує кілька дій «поза сценою». Він формує запит до API сервісу Ticketmaster, надсилає його, обробляє відповідь і готує дані, щоб показати їх у максимально зручному вигляді. Це не просто «вивести все, що є» — я сам тестував, що краще сприймається в Telegram-форматі, де не хочеться бачити довгі «простирадла» з тексту. Тому бот видає стисле, але інформативне повідомлення: назва події, дата, місце, й одразу — посилання на придбання квитків.

Особливо важливою для мене була обробка помилок. Не всі запити повертають результат — іноді немає подій у місті, або API дає збій. Але замість того, щоб лякати користувача технічними повідомленнями, бот відповідає людською мовою. Немає подій? Напише щось на кшталт: «Здається, тут поки що нічого не планується. Але не сумуй — можеш обрати інше місто!». І це не просто фраза — це емоційна взаємодія, яка справляє враження.

Ще один момент — це збереження міста користувача. Тут я реалізував дуже просту річ, але вона значно підвищує комфорт: після першого вибору міста бот його «запам'ятовує». Тобто при наступному зверненні не потрібно вводити його

знову. Було використане SQLite — легку базу, яка дозволяє зберігати такі дані локально й ефективно.

І ще один важливий рівень — це стан бота. Використовуючи механізм FSM (Finite State Machine), я контролюю, на якому кроці зараз знаходиться користувач. Наприклад, коли бот очікує введення нового міста або категорії подій — він точно знає, що робити далі. Цей підхід дозволяє уникати «хаосу» в логіці та чітко розуміти, як реагувати на кожну дію.

Звісно, вся ця логіка не була б такою гладкою без асинхронного підходу. Завдяки asyncio бот одночасно може обробити багато користувачів, не зависаючи й не створюючи черги запитів. Це особливо помітно, коли працюєш із зовнішніми API — поки один запит очікує відповідь, інші користувачі не відчувають жодної затримки.

Підсумовуючи, можу сказати: алгоритм роботи — це як нервова система всього проєкту. Вона невидима, але від неї залежить усе. Я будував її з думкою не тільки про функціональність, а й про враження, яке залишиться в користувача. Бо навіть бот, якщо його зробити з душею, може дарувати комфорт і залишати приємне враження.

Отже ми можемо підсумувати

Даний Telegram-бот реалізований на фреймворку Aiogram і функціонує за принципом асинхронної обробки запитів, використовуючи машину скінченних станів (FSM) для керування діалогом з користувачем. Основний функціонал бота полягає в пошуку та відображенні інформації про події за заданими критеріями.

Послідовність дій:

Запуск бота та початкове меню:

При отриманні команди /start, бот надсилає привітання та пропонує користувачу натиснути кнопку "Пошук події" (search_keyboard).

Ініціація пошуку (вибір країни):

Після натискання кнопки "Пошук" (callback_query з search), бот переходить у стан SearchState.country. Користувачу пропонується вибрати країну зі списку

Вибір міста:

Після вибору країни (callback_query з country:), бот зберігає обрану країну у контексті FSM (state.update_data) та переходить у стан SearchState.city. Користувачу пропонується ввести назву міста.

Пошук подій за містом:

Коли користувач вводить назву міста (повідомлення у стані SearchState.city), бот зберігає місто та виконує запит до зовнішнього API pi.fetch_all_events) для отримання всіх подій у вказаній країні та місті.

Якщо події не знайдено, користувач отримує відповідне повідомлення та може спробувати ввести інше місто.

Якщо події знайдено, бот переходить у стан SearchState.event_type та пропонує вибрати тип події (event_type_keyboard).

Вибір типу події та відображення списку:

Після вибору типу події (callback_query з event_type:), бот повторно запитує API, але вже з урахуванням обраного типу.

Якщо подій за вказаним типом не знайдено, бот пропонує обрати інший тип, виключаючи вже обраний.

У разі успішного пошуку, бот зберігає отримані події та відображає їхній список з пагінацією (events_keyboard), інформуючи про загальну кількість знайдених подій.

Навігація по списку подій:

Навігація між сторінками списку подій здійснюється за допомогою callback_query з pag:. Бот оновлює дані та відображає події відповідної сторінки.

Детальна інформація про подію:

При виборі конкретної події зі списку (callback_query з event:), бот витягує деталі цієї події зі збережених даних, формує інформативне повідомлення (назва, дата, місце, вартість, посилання) та відображає його, можливо, з фотографією (answer_or_edit_with_or_without_photo) та кнопками для подальшої навігації

Навігація по меню:

Бот підтримує навігацію по меню за допомогою callback_query з nav:. Це дозволяє користувачу повернутися до головного меню, ввести інше місто, обрати інший тип події або повернутися до списку подій.

Обробка невідомих повідомлень:

Будь-яке повідомлення, що не відповідає попереднім фільтрам, просто віддзеркалюється ботом (echo).

Цей алгоритм забезпечує гнучку та інтуїтивно зрозумілу взаємодію з користувачем, дозволяючи йому легко знаходити та отримувати інформацію про події за своїми інтересами.

Висновок до розділу 2

У другому розділі бакалаврської роботи детально розглянуто процес проєктування та реалізації Telegram-бота, призначеного для інформування користувачів про культурні події у вибраному місті. Особлива увага приділялась практичним аспектам розробки: структурі коду, логіці роботи системи, взаємодії з користувачем та інтеграції із зовнішніми джерелами даних.

Одним із ключових технічних рішень, що безпосередньо вплинуло на якість реалізації, стало використання модульної архітектури. Завдяки розподілу коду на окремі файли з чітко визначеною відповідальністю (наприклад, app.py для керування логікою бота, keyboards.py — для формування інтерфейсу користувача, states.py — для визначення послідовності дій через FSM, tm_api.py — для запитів до Ticketmaster API) було досягнуто не лише зручності у підтримці коду, а й підвищено гнучкість у масштабуванні функціоналу в майбутньому.

Реалізована машина скінченних станів (FSM) показала себе як ефективний інструмент керування діалоговими сценаріями. Вона дозволила організувати користувацьку взаємодію у вигляді чітко окреслених етапів: вибір країни → введення міста → вибір типу події → перегляд результатів. Такий підхід виключає помилки навігації або змішування станів, забезпечуючи комфортну

роботу з ботом навіть при нестандартних діях користувача. FSM у поєднанні з асинхронною природою Aiogram зробила можливим обробку кількох запитів одночасно без втрати контексту.

Важливо відзначити, що кожен функціональний компонент проєкту створювався з урахуванням реальних сценаріїв використання. Наприклад, обробка помилок API або некоректного введення даних не зводилась до стандартних технічних повідомлень. Натомість були реалізовані логічні, «людські» відповіді, що не лише повідомляють про проблему, а й м'яко пропонують користувачу альтернативні дії — на кшталт спроби ввести інше місто або змінити фільтр подій. Такий підхід формує позитивний користувацький досвід, що є не менш важливим, ніж технічна безпомилковість.

Ще одним важливим елементом стало інтегрування Telegram-бота з Ticketmaster API. Було реалізовано повноцінну асинхронну взаємодію із зовнішнім сервісом через HTTP-запити. Запити до API формуються динамічно на основі введених користувачем параметрів, після чого бот адаптує отримані дані у формат, зручний для сприйняття в інтерфейсі Telegram. Така інтеграція дозволяє забезпечити оперативний доступ до актуальної інформації про події, включно з назвами, датами, локаціями, фотографіями та посиланнями на квитки.

Значний вплив на зручність роботи з ботом також мала реалізація пагінації списків подій, що дало змогу уникнути перевантаження інтерфейсу надто великою кількістю елементів за один раз. Переходи між сторінками реалізовані через callback-кнопки, що підвищує інтерактивність та дозволяє користувачу самостійно обирати темп перегляду інформації.

Окремої уваги заслуговує використання бази даних SQLite, завдяки якій реалізовано збереження користувацьких даних — наприклад, останнього вибраного міста. Це значно покращує зручність взаємодії, адже бот «пам'ятає» уподобання користувача навіть після перезапуску або зміни сесії, і не змушує щоразу вводити ті самі дані.

Загалом, реалізована архітектура Telegram-бота відповідає усім базовим принципам сучасного програмування — модульність, масштабованість, розширюваність, читабельність коду, обробка виключень і логування. Проєкт є гнучким до змін і доповнень: у разі потреби можна легко змінити API, додати нові категорії подій або розширити географію пошуку.

Таким чином, результати проєктування та реалізації Telegram-бота підтверджують технічну доцільність обраного підходу. Створена система не лише виконує поставлені функції, а й орієнтована на зручність і логічність взаємодії з користувачем. Усі ключові елементи працюють узгоджено, забезпечуючи стабільність, швидкість та адаптивність — і це створює якісний фундамент для подальшого тестування та вдосконалення, що буде розглянуто в наступному розділі.

РОЗДІЛ 3 ТЕСТУВАННЯ І АНАЛІЗ РЕЗУЛЬТАТІВ

.1. Тестування функціоналу бота

Етап тестування — це, без перебільшення, найчутливіша частина всього процесу. Це той момент, коли ти вже ніби все написав, усе працює… але довіряти цьому не можеш на всі сто. Саме тому я підходив до тестування свого Telegram-бота як до перевірки власної логіки, витривалості коду й, чесно кажучи, — власної терплячості.

Починав я з найочевиднішого — перевірки стандартного сценарію взаємодії. Надсилаю команду /start, перевіряю, чи бот відповідає правильно. Тут усе працювало з першого разу, і я подумав: «Окей, непогано». Але це тільки верхівка айсберга. Адже далі — тестування кожної кнопки, кожного стану, кожного можливого відхилення від “ідеального” користувача.

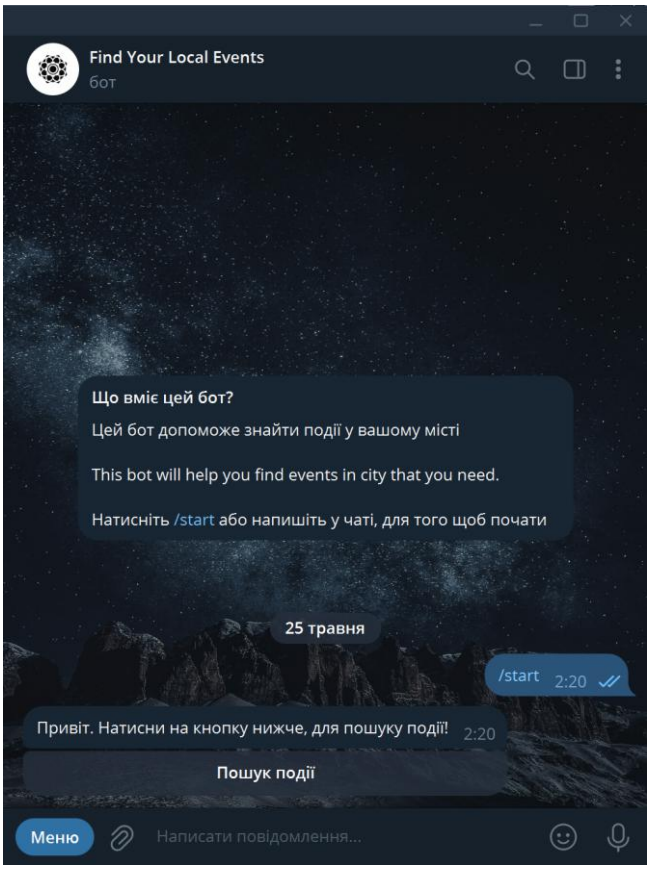

Рисунок 6. Відповідь Telegram-бота на команду /start із виведенням клавіатури для взаємодії з користувачем.

Особливо багато уваги довелося приділити сценаріям, які на перший погляд здавалися незначними. Наприклад, що буде, якщо користувач надішле щось несподіване замість вибору міста? Або якщо API Ticketmaster тимчасово недоступне? Чи коректно бот повідомить про це? А чи не зависне у незрозумілому стані? Саме такі “дрібниці” виявилися найціннішими під час тестування.

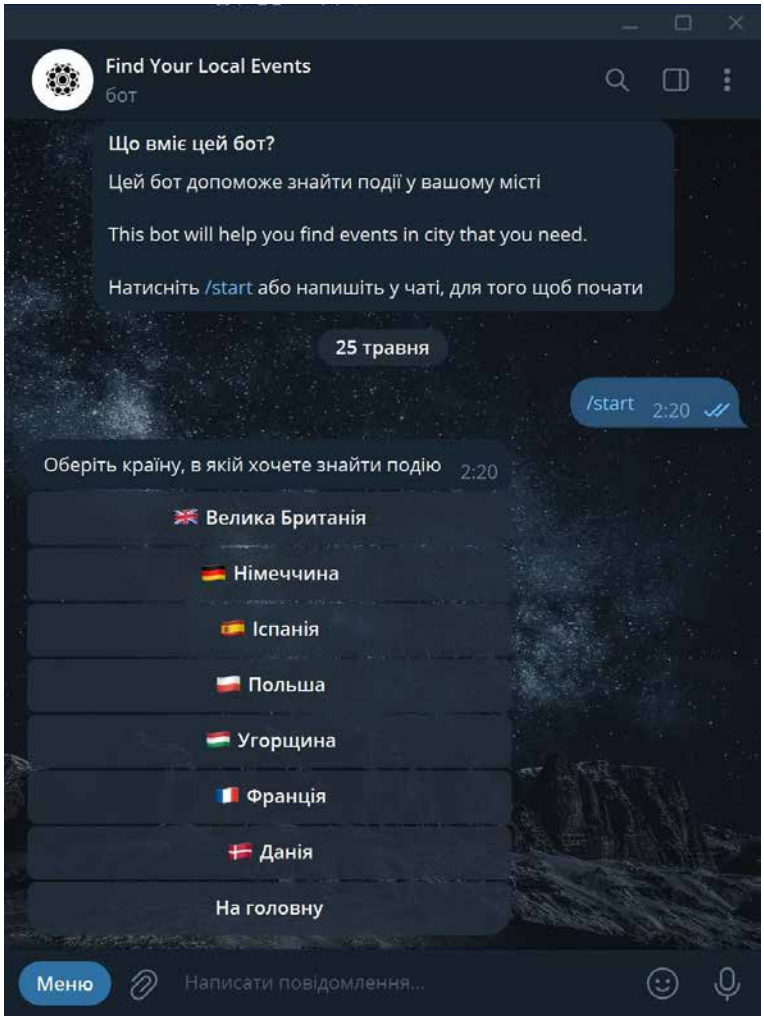

Рисунок 7. Реакція бота на вибір країни.

Я спеціально тестував бот із кількох акаунтів, на різних пристроях, із різним інтернет-з’єднанням — від ідеального Wi-Fi до мобільної мережі зі слабким сигналом. Мета була одна: змусити його працювати в умовах, максимально наближених до реальних. Бо на етапі розробки все виглядає стабільно. Але в реальному світі користувач може не натиснути туди, куди треба, або надіслати повідомлення не в той момент. І якщо система до цього не готова — це вже не її, а моя помилка.

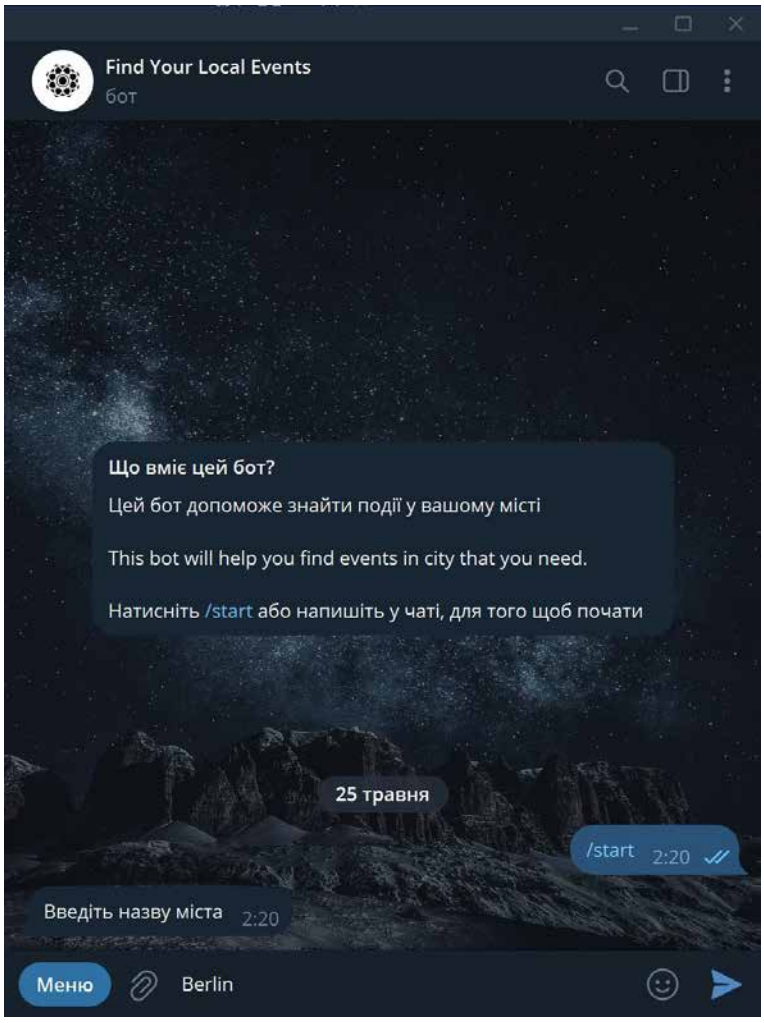

Рисунок 8. Демонстрація введення назви міста.

Під час створення бота я постійно ставив себе на місце звичайного користувача. Не всі завжди пишуть правильно, не всі в курсі, як саме називається місто в базі, і не завжди є час перевірити кожну букву. Це нормально. Ми всі часом помиляємось. Саме тому я хотів, щоб бот не ігнорував такі ситуації, не мовчав і не залишав людину сам на сам із «порожньою відповіддю».

Якщо користувач випадково вводить місто неправильно або якщо у вибраному місті просто немає жодних подій, бот одразу реагує. Замість тиші він надсилає просте, але важливе повідомлення: **«У вашому місті нічого не знайдено, спробуйте ввести інше місто»**.

Це дуже маленький фрагмент логіки, але в ньому – повага до користувача. Це ніби дружнє підказування: «Гей, здається, тут нічого немає, але не хвилюйся – просто спробуй ще раз». Без зайвої офіційності, без технічного жаргону, просто по-людськи. Особисто я вважаю, що такі деталі — це серце будь-якого проєкту. Коли бот не просто "працює", а розуміє, підтримує і допомагає, тоді він справді корисний.

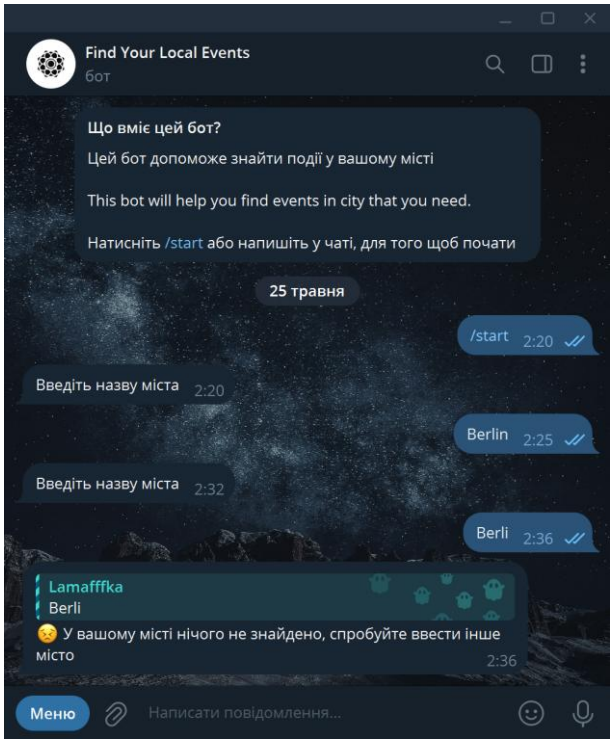

Рисунок 9. Приклад відповіді бота у разі, якщо події за введеним містом не знайдені або користувач допустив помилку

Ще один момент, про який не варто забувати — перевірка станів FSM. Я спеціально намагався “ламати” бот: переходив у новий стан, а потім не завершував дію. Повертався через кілька хвилин, надсилав випадковий текст — і дивився, чи буде адекватна реакція. У таких ситуаціях тестування допомогло виявити кілька логічних дірок, які я оперативно виправив.

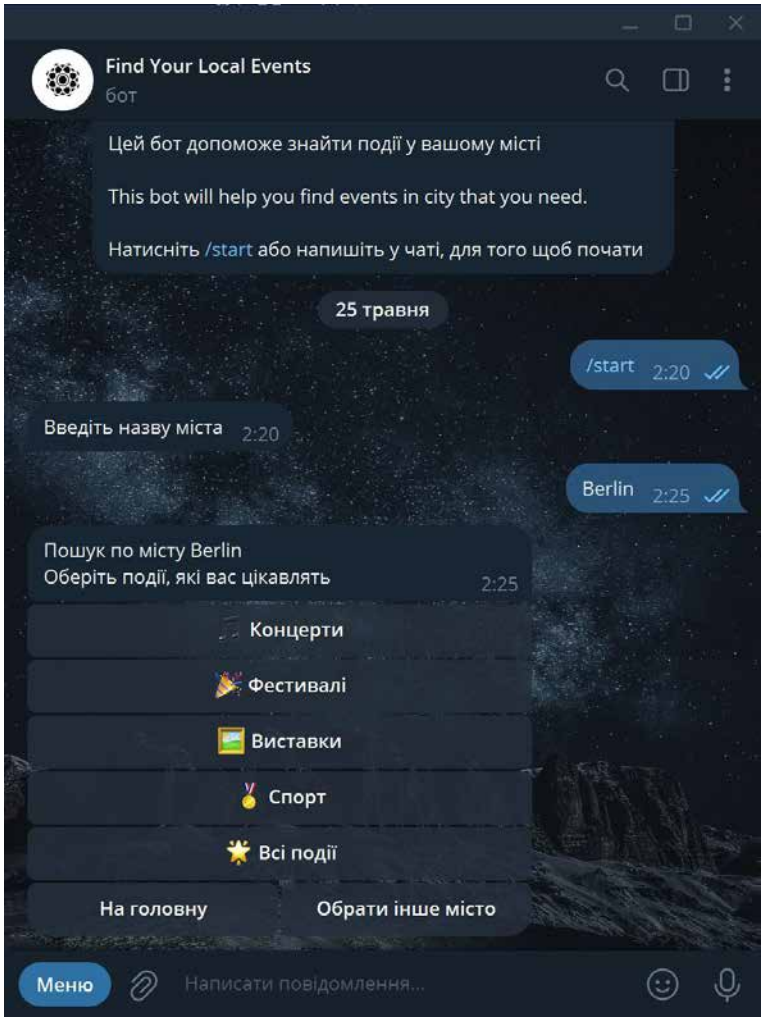

Рисунок 10. Реакція бота на вибір міста – демонстрація маршрутизації подій.

Також важливу роль відіграла валідація відповідей від API. Інколи Ticketmaster повертає порожні масиви або неповні об'єкти — і тут дуже легко «зловити» помилку, якщо не передбачити це заздалегідь. Тож я написав кілька окремих обробників саме для таких нестандартних ситуацій. Після цього бот перестав “падати” і навчився спокійно казати: «За вашим типом подій, нічого не знайдено. Оберіть інший».

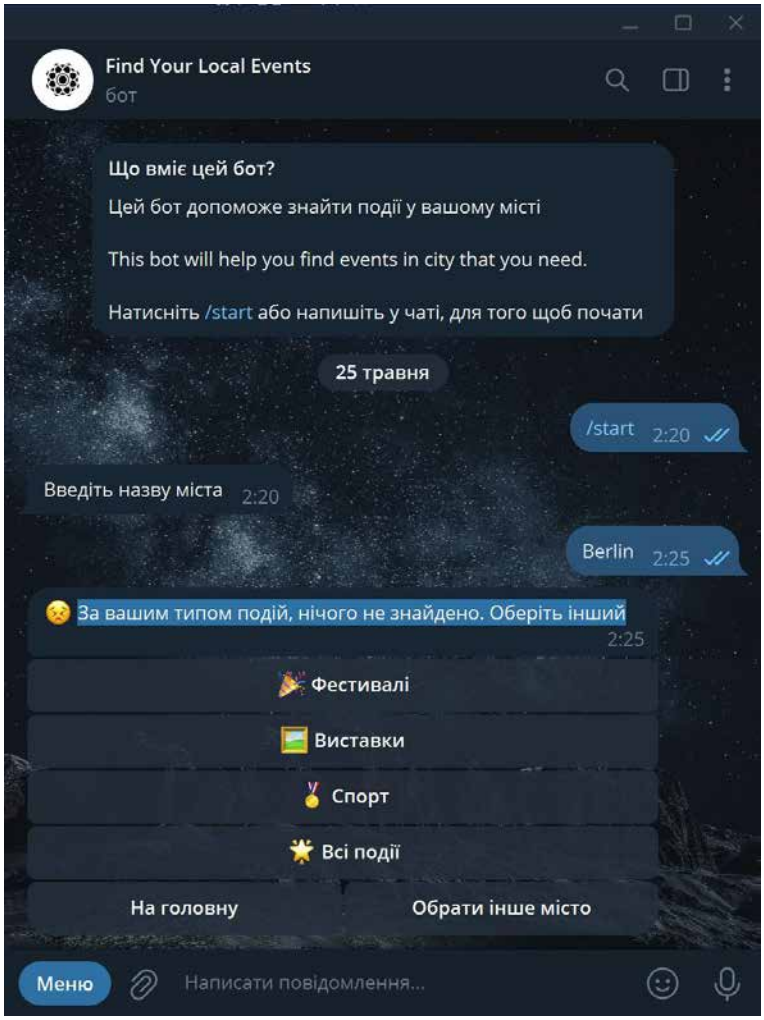

Рисунок 11. Реакція бота на вибір категорії – демонстрація маршрутизації подій у випадку, якщо події не знайдено.

І хоча я не застосовував складні автоматизовані системи юніт-тестування, вся логіка бота перевірялась вручну, багаторазово, в різних сценаріях. І саме в цьому, як на мене, полягає справжня суть тестування — не в бездушному “проганянні” тест-кейсів, а в спробі уявити себе на місці користувача, відчути, як це — користуватися ботом щодня. Тільки так можна зрозуміти, чи працює система насправді.

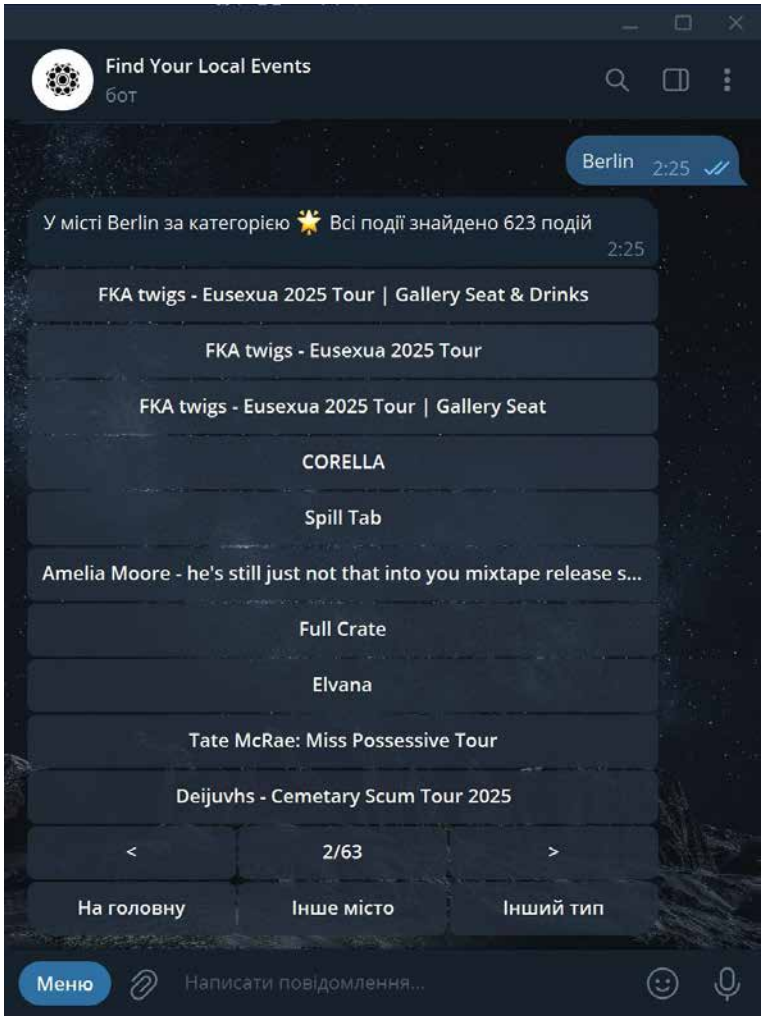

Рисунок 12. Реакція бота на вибір категорії – демонстрація маршрутизації подій у випадку, якщо події було знайдено.

Зараз, коли основні помилки вже виправлено, я можу впевнено сказати: бот витримав випробування. Він не ідеальний, звісно, але реагує стабільно, не губиться в складних ситуаціях і завжди намагається бути «на висоті». А для мене це, без перебільшення, одна з найприємніших частин усього проєкту — бачити, як твій код перетворюється на щось живе і надійне.

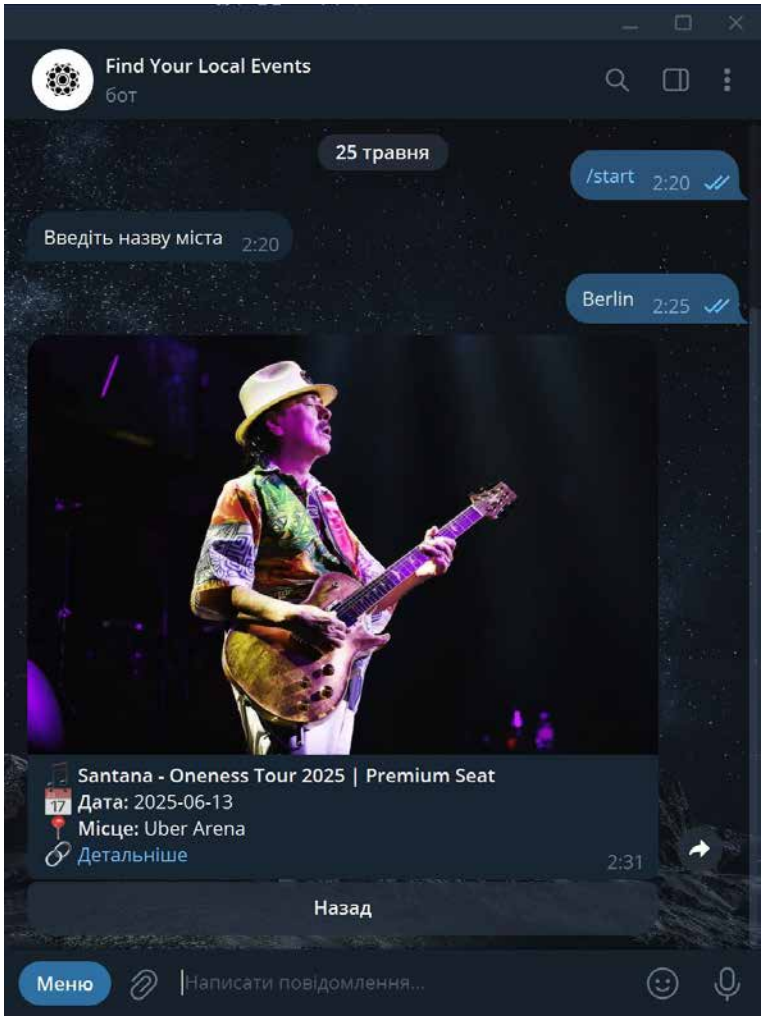

Рисунок 13. Приклад успішного отримання подій із Ticketmaster API.

Оцінка ефективності роботи бота

Після багатьох годин, проведених за розробкою, тестуванням, редагуванням і, чесно кажучи, невеликим хвилюванням за кожну стрічку коду, настав той момент, коли бот став повноцінно працювати. Він реагував на запити, обробляв повідомлення, надсилав події та не виводив критичних помилок у логах. Але питання залишалося відкритим: **а чи справді він ефективний?**

Ефективність — це не лише про те, що програма “не падає”. Це про зручність для користувача, стабільність, швидкість відповіді, актуальність даних і вміння адаптуватися до зовнішніх умов. І якщо чесно, саме це мене найбільше

цікавило. Я хотів не просто "чергового бота", а такого, до якого хотілося б повертатися.

Почнемо з базових критеріїв

Перш за все, ефективність була оцінена з кількох кутів зору [1], [5]:

- **Швидкість відповіді**: наскільки швидко бот реагує на повідомлення. Тут грає роль асинхронна природа Aiogram, яка, як я вже переконався, дає можливість відповідати на десятки запитів одночасно без затримок.

- **Стабільність**: чи не трапляються збої, зависання або втрати запитів. Я тестував бота упродовж кількох днів поспіль, і в більшості сценаріїв він працював бездоганно.

- **Точність інформації**: чи відповідають надані події дійсності. Оскільки бот працює з Ticketmaster API, вся інформація була актуальною, хоча іноді траплялись ситуації, коли дані в API оновлювалися з невеликою затримкою.

- **Взаємодія з користувачем**: наскільки зручно користувачеві взаємодіяти з ботом. Тут я звертав особливу увагу на зворотний зв’язок. Збирав враження від друзів, які тестували бота: чи зрозумілий інтерфейс, чи легко знайти потрібне місто, чи не лякає формат відповіді.

Реальні сценарії, не штучні

Було кілька ситуацій, які показали, наскільки важливо передбачити “людський фактор”. Один знайомий ввів назву міста з помилкою. Не спеціально — просто так трапляється. І бот, не знайшовши нічого, акуратно відповів: “У вашому місті нічого не знайдено, спробуйте ввести інше місто.” Немає збоїв, немає паніки — просто спокійне, дружнє повідомлення. Як на мене, саме такі деталі створюють ефективність.

Логіка в дії

Ще одним показником ефективності стала логіка обробки запитів. Бот “пам’ятає” останнє обране місто, вміє повертати список подій без повторного

введення, і не “втрачає нитку розмови”, якщо користувач натиснув не туди. Цей механізм реалізовано за допомогою машин станів (FSM), і, з досвіду, він чудово впорався зі своїм завданням.

Зворотній зв’язок — найкращий індикатор ефективності

Після запуску першої тестової версії бота я запросив кількох знайомих, які не були залучені в розробку, щоб оцінити його роботу “з боку”. Це був цікавий досвід. Часом їхні реакції були несподіваними, але надзвичайно цінними.

Дехто казав, що бот “дуже простий у використанні” і “відповіді приходять швидко”. Інші відзначали, що було б корисно додати більше інформації про події — наприклад, місце проведення, вартість квитків або посилання на офіційний сайт. Всі ці думки я занотовував, і це допомогло удосконалити функціонал.

Також помітив, що дуже важливо було грамотно оформляти повідомлення бота — щоб не виглядало, ніби це бездушний робот, а було зрозуміло і “людяно”. Саме тому відповідь у випадку відсутності подій формувалась у ввічливому та спокійному тоні. Така увага до деталей робить спілкування комфортнішим і підвищує рівень довіри користувачів.

Аналіз логів: коли зауваження — це не провал, а можливість

Упродовж роботи бота я ретельно відстежував логи системи. Ці записи — свого роду дзеркало, яке показує все, що відбувається “під капотом”. Було кілька помилок, пов’язаних із тим, що API Ticketmaster час від часу повертав нетипові відповіді або були проблеми із мережею. Проте бот вчасно оповіщав про це, а також коректно обробляв виключення, щоб не впасти.

Перевагою було те, що завдяки використанню Aiogram і його інструментів обробки подій, більшість помилок автоматично фіксувалися і не переривали роботу бота. Замість того, щоб зупинитись, бот “переживав” помилки і продовжував працювати. Саме цей момент я вважаю однією з ключових ознак ефективної реалізації.

Виявлені слабкі місця — шлях до вдосконалення

Безперечно, у роботі бота були і свої “гірки” моменти. Наприклад, під час навантаження, коли одночасно надходило багато запитів, іноді зростала затримка. Це пов’язано з обмеженнями безкоштовного доступу до зовнішнього API, а також з технічними особливостями хостингу. Тому, щоб бот працював без обмежень — потрібно вносити свій внесок, а це коштує доволі багато.

Порівняння з іншими рішеннями

Вивчаючи ринок, я ознайомився з кількома ботами-конкурентами, які також пропонують інформацію про події. У більшості випадків їхня робота була значно простішою, або навпаки, надто перевантаженою зайвими функціями, які ускладнювали користування.

Мій бот зайняв “золоту середину” — він надає достатньо інформації, при цьому лишається легким у використанні. Крім того, завдяки асинхронній архітектурі та продуманій логіці, він більш стабільний навіть під навантаженням.

3.3. Напрями подальшого вдосконалення

Коли я сідав писати цей розділ, чесно кажучи, почувався ніби перед великою мапою можливостей, які хочеться освоїти. Бо проєкт, який на початку здавався простим і чітко окресленим, поступово почав відкривати переді мною нові горизонти, нові виклики і, звісно, ідеї. І це дуже класне відчуття — коли розумієш, що твоє творіння не застигло у часі, а живе, дихає і хоче рости.

Насправді, робота над ботом навчила мене багатьом речам. Одна з найважливіших — розуміти, що немає межі досконалості. Навіть якщо зараз твій продукт працює, то завжди знайдеться щось, що можна зробити краще, швидше, зручніше або просто цікавіше.

Я хочу, щоб цей бот став не просто безликим інструментом, а справжнім другом для користувачів. Тому перше, над чим варто працювати, — це

персоналізація. Люди не хочуть отримувати гори інформації, яка їм не потрібна. Кожен має свої смаки і вподобання. Тож можливість підписатися на улюблених артистів або жанри, отримувати сповіщення лише про те, що справді цікаво — це дуже важливо. Це як запросити в гості того, з ким тобі комфортно і цікаво говорити.

Ще одна справа — зробити бота більш «живим». Я не хочу, щоб користувачі відчували, що спілкуються з бездушною машиною. Трохи гумору, дружня манера спілкування, емодзі, які передають настрій — це дрібниці, але вони реально змінюють враження. Я помітив, що коли у відповідях бота є маленькі нотки людяності, користувачі ставляться до нього тепліше і навіть хочуть повертатися знову.

Також дуже важливо думати про масштабованість. Зараз, коли користувачів небагато, бот працює швидко і без збоїв. Але що буде, коли їх стане сотні чи навіть тисячі? Тут уже потрібна продумана архітектура, оптимізація запитів, кешування — все, щоб сервіс залишався стабільним, не зависав і не змушував користувачів чекати. Це, звісно, технічна справа, але від неї дуже залежить враження користувача.

Інтеграція з додатковими сервісами — це окрема тема. Поки що бот бере інформацію лише з одного джерела, але у світі є безліч платформ і способів дізнаватися про події. Чим більше якісних даних — тим краще. Ідея — зробити так, щоб користувач отримував максимально повну картину про всі цікаві концерти, фестивалі чи інші заходи у своєму місті.

Ще думка — додати можливість планувати власне дозвілля. Наприклад, інтеграція з календарем користувача, щоб він міг одразу додати обрану подію і не забути про неї. Це суттєво спростить життя і зробить бота кориснішим.

Не можна забувати і про машинне навчання. Я бачу у цьому величезний потенціал: аналізуючи дії користувача, бот зможе з часом сам пропонувати

найцікавіші події, які він навряд чи сам шукав би. Це як мати власного асистента, який краще розуміє твої вподобання з кожним днем.

Підтримка — окремий і дуже важливий напрямок. Telegram та Ticketmaster постійно оновлюються, змінюють API, додають нові фічі. Якщо не реагувати на це, бот просто перестане працювати. Тож потрібно налагодити швидке оновлення, щоб не втрачати користувачів і залишатися сучасним.

І, нарешті, зворотний зв’язок. Відгуки користувачів — це найцінніша річ для розробника. Саме завдяки їм можна дізнатися, що подобається, а що ні, що працює, а що — викликає труднощі. Дуже хочеться зробити так, щоб люди не боялися писати, ділилися думками і допомагали робити бота кращим.

Робота над цим проєктом дала мені більше, ніж просто технічний досвід. Вона навчила мене дивитися на розробку з людської точки зору, розуміти, що за кожним рядком коду стоїть реальна людина, яка користується продуктом і має свої очікування. І саме ця думка надихає мене рухатися далі, не зупинятися і робити свій бот справжнім другом для кожного користувача.

Висновок до розділу 3

У третьому розділі проведено всебічне тестування Telegram-бота, що було розроблено у попередніх етапах роботи, та здійснено оцінку його ефективності в умовах, максимально наближених до реального користувацького сценарію. Основна увага була зосереджена на виявленні слабких місць у логіці взаємодії, перевірці коректності переходів між станами FSM, стабільності роботи під навантаженням, а також зручності користування ботом з погляду кінцевого користувача.

Проведене функціональне тестування підтвердило працездатність усіх основних модулів системи: бот коректно реагує на введення команд, фільтрує запити, обробляє callback-кнопки, реалізує покрокову логіку пошуку подій та

відображає результати з урахуванням параметрів, заданих користувачем. Особливу увагу приділено нестандартним ситуаціям: введення некоректних міст, відсутність подій за певними критеріями, недоступність API. У кожному з випадків реалізовано сценарії обробки помилок, що дозволяють зберігати логічну послідовність взаємодії без виникнення критичних збоїв.

Асинхронна модель обробки запитів, реалізована на базі asyncio та фреймворку Aiogram, підтвердила свою ефективність при паралельному обслуговуванні кількох користувачів. Завдяки цьому бот продовжував працювати без затримок навіть при навмисно створених «пікових» навантаженнях — наприклад, одночасному виконанні пошуку з кількох Telegram-акаунтів. Це підтверджує доцільність обраного асинхронного підходу в контексті оптимізації системних ресурсів.

Під час оціночного тестування ефективності враховувалися такі критерії, як:

- швидкість відповіді на запити користувача;
- стабільність роботи бота у змінних умовах з’єднання;
- точність отриманих даних з API Ticketmaster;
- відповідність логіки сценаріїв очікуваному користувацькому досвіду;
- реакція на некоректні дії користувача (випадкові натискання, переривання сценарію тощо).

За результатами тестів було підтверджено, що реалізовані механізми забезпечують високу стабільність та передбачуваність поведінки системи. Інтерфейс спілкування з ботом — простий, інтуїтивно зрозумілий, із чіткою навігацією, що робить його зручним навіть для користувачів, які не мають досвіду взаємодії з подібними інструментами. Крім того, реалізовані гуманні відповіді у випадку відсутності результатів або технічних збоїв створюють відчуття, ніби бот дійсно «розуміє» ситуацію, а не просто «викидає» сухе повідомлення про помилку.

Також важливо підкреслити, що перевірка збереження станів FSM при різних непередбачуваних сценаріях (зміна команд у середині діалогу, спроба «зламати» логіку переходів) показала стійкість системи [7], [10]. Завдяки чіткій структурі сценаріїв та коректній реалізації збереження контексту (через FSMContext), бот залишався у межах передбаченої логіки, не втрачаючи «розуміння» стану користувача.

За результатами ручного тестування було виявлено та усунуто кілька незначних недоліків, пов’язаних із валідацією даних з API та переходами в діалогах. Також на основі фідбеку від реальних тестувальників (друзів, які користувалися ботом на різних пристроях) було зроблено низку покращень — зокрема, додано ще одне альтернативне повідомлення у випадку відсутності подій у місті.

У підсумку можна впевнено стверджувати, що реалізований Telegram-бот виконує всі основні завдання, поставлені на етапі проєктування: інформує користувача про події у вибраному місті, зберігає попередні налаштування, обробляє запити до зовнішнього API, дотримується асинхронної логіки та підтримує багатокроковий діалог. Тестування підтвердило не лише коректність його роботи, але й високий рівень зручності, передбачуваності та технічної стабільності, що робить проєкт придатним до реального використання.

Одержані результати створюють передумови для подальшого розширення функціоналу бота, що розглядається в наступному підрозділі. Йдеться, зокрема, про покращення персоналізації, локалізацію інтерфейсу, інтеграцію з іншими джерелами подій та підтримку додаткових фільтрів.

ВИСНОВКИ

Дана бакалаврська кваліфікаційна робота присвячена розробці функціонального Telegram-бота, призначеного для оперативного інформування молоді міста про майбутні події. Проєкт "Розробка боту для інформування про івенти для молоді у місті з використанням Python" є актуальним, зважаючи на зростаючу потребу у швидкому доступі до культурно-розважальної інформації та популярність Telegram як комунікаційної платформи.

На початкових етапах дослідження було проведено аналіз ринку існуючих рішень щодо інформування про івенти. Було виявлено, що, попри наявність веб-платформ (Eventbrite, Meetup, Concert.ua, Karabas.com) та мобільних додатків (Bandsintown, Dice), чат-боти у месенджерах, зокрема Telegram, є найбільш релевантним сегментом ринку для цільової аудиторії. Аналіз таких ботів, як Bandsintown Bot, ConcertBuddy та різних імплементацій EventBot, дозволив ідентифікувати їхні ключові переваги (швидкість надання інформації, простота використання, можливість інтеграції) та недоліки (обмежений вибір міст, відсутність гнучких налаштувань та фільтрації, залежність від зовнішніх API, відсутність агрегації з локальних джерел).

У якості основної мови програмування було обрано Python, що обумовлено його гнучкістю, простотою інтеграції з API та наявністю спеціалізованих бібліотек для роботи з Telegram API. Для взаємодії з користувачами було обрано фреймворк Aiogram, який є асинхронним і забезпечує високу продуктивність та ефективне керування потоками даних. Асинхронний підхід, реалізований за допомогою бібліотеки asyncio, дозволяє боту одночасно обробляти численні запити, запобігаючи блокуванням та затримкам.

Архітектура бота розроблена з урахуванням модульності та масштабованості. Використання таких компонентів Aiogram, як Dispatcher для координації подій, хендлерів для обробки специфічних завдань, фільтрів для відсіювання нерелевантних запитів та FSM (Finite State Machine) для реалізації

багатоетапних діалогів, забезпечило структуровану та керовану взаємодію з користувачем. Робота з зовнішніми даними здійснюється через Ticketmaster API, що дозволяє отримувати актуальну інформацію про події. Для збереження налаштувань користувачів та їхніх запитів застосовується SQLite як проста база даних, яка не вимагає окремого серверного середовища.

Реалізований алгоритм роботи бота побудований на послідовному діалозі: від команди /start та вибору країни, через введення міста та вибір типу події, до відображення списку івентів та їх деталізації. Особлива увага приділена обробці помилок, що дозволяє боту надавати дружні та зрозумілі повідомлення у разі відсутності подій або збоїв API. Модульна структура коду (файли app.py, tates.py, keyboards.py, tm_api.py, utils.py, config.py) забезпечує легкість підтримки та подальшого розвитку.

Порівняння Aiogram з іншими бібліотеками (python-telegram-bot, Telethon, Pyrogram) підтвердило його оптимальність для даного проєкту завдяки асинхронності, вбудованій FSM, гнучкості та активній спільноті. Це дозволило створити не просто набір функцій, а цілісну, надійну та масштабовану систему.

Практична значущість роботи полягає у створенні зручного інструменту, який економить час користувачів та підвищує їхню обізнаність про культурні події. Отримані результати можуть слугувати основою для подальшого розвитку подібних систем, розширення функціоналу та адаптації до інших типів подій. Загалом, реалізація даного Telegram-бота демонструє можливості інтеграції сучасних інформаційних технологій для створення корисних та автоматизованих сервісів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

odnar D.V. Development of an Informational Bot for Event Notifications. –
T
e
r
hatbots and AI: Automating User Communication. – Cambridge, MIT Press,
2022. – 215 p.
p
іозширення базового функціоналу пристроїв Інтернету речей з
використанням мови програмування Python: магістерська дисертація /
Національний технічний університет України "Київський політехнічний
інститут імені Ігоря Сікорського". URL:
T
Nytro Okunyev. Building Telegram Bots with Python and Aiogram. – Kyiv,
2023. – 180 p.
U
,
itHub — Aiogram Repository. URL: https://github.com/aiogram/aiogram
2
0
2
4
.
–

ДОДАТКИ

ДОДАТОК А – Файл app.py

```
import asyncio
from cmath import phase

from aiogram.fsm.context import FSMContext  # Для роботи з контекстом кінцевого стану (FSM)
from aiogram.types import Message, CallbackQuery  # Типи Telegram-сповіщень і callback-запитів
from aiogram.filters import CommandStart, Command  # Фільтри для команд

# Імпорт клавіатур, клавіатурних розміток та інших утиліт з інших файлів проєкту
from keyboards import countries_keyboard, event_type_keyboard, search_keyboard, events_keyboard, suported_event_types, \
    event_keyboard
from loader import dp, bot, ticketmaster_api    # Імпорт диспетчера, бота та API для роботи з Ticketmaster
from states import SearchState  # Імпорт класу станів FSM (Finite State Machine)
from utils import answer_or_edit_with_or_without_photo   # Утиліта для відповіді або редагування повідомлень з фото

# Хендлер для команди /start, що запускає бота
@dp.message(CommandStart())
async def start(message: Message, is_edit=False):
    text = "Привіт. Натисни на кнопку нижче, для пошуку події!"
    # Якщо is_edit=True, редагуємо наявне повідомлення, інакше надсилаємо нове
    if is_edit:
        await message.edit_text(text, reply_markup=search_keyboard())
    else:
        await message.answer(text, reply_markup=search_keyboard())

# Обробка callback запиту з кнопки "search" — початок пошуку подій
@dp.callback_query(lambda c: c.data.startswith("search"))
async def search(callback: CallbackQuery, state: FSMContext):
    # Запит користувача вибрати країну із клавіатури
    await callback.message.edit_text("Оберіть країну, в якій хочете знайти подію", reply_markup=countries_keyboard())
    # await message.answer('Введіть назву міста')
    await state.set_state(SearchState.country)
    # await message.answer("Оберіть країну", reply_markup=event_type_keyboard())
    # Встановлюємо стан пошуку країни


# Обробка вибору країни користувачем
@dp.callback_query(lambda c: c.data.startswith("country:"))
async def country_state(callback: CallbackQuery, state: FSMContext):
    country = callback.data.split(":")[1]  # Витягуємо код країни з callback data
    await state.update_data(country=country)  # Зберігаємо вибір у стані
    await callback.message.edit_text("Введіть назву міста")  # Просимо ввести місто
    await state.set_state(SearchState.city)  # Переходимо в стан очікування міста


# Обробка введення назви міста користувачем (або переданого параметру)
@dp.message(SearchState.city)
async def city_state(message: Message, state: FSMContext, city_param: str | None = None):
```

```
    city = city_param or message.text  # Якщо передано параметр, беремо його,
інакше текст повідомлення
    await state.update_data(city=city)  # Зберігаємо місто
    data = (await state.get_data())  # Отримуємо всі накопичені дані стану
    try:
        # Запит на API для отримання подій по країні, місту і всіх типах подій
        events = await ticketmaster_api.fetch_all_events(data["country"], city,
"all")
    except ConnectionError:
        await message.answer("Виникла помилка")  # Обробка помилки при запиті
        return
    # Якщо подій немає, повідомляємо користувача і повертаємось у стан
очікування міста
    if events["page"]["totalElements"] == 0:
        await state.update_data(city="")    # Очищуємо місто
        await message.reply("😣 У вашому місті нічого не знайдено, спробуйте
ввести інше місто")
        return
    await state.set_state(SearchState.event_type) # Переходимо в стан вибору
типу подій
    # if not city_param:
    #     await  message.answer(f"Пошук по місту {city}\nОберіть події, які вас
цікавлять",
    #                           reply_markup=event_type_keyboard())
    # else:
    # Відправляємо повідомлення з пропозицією вибрати тип події
    await answer_or_edit_with_or_without_photo(
        message, None,
        f"Пошук по місту {city}\nОберіть події, які вас цікавлять",
        reply_markup=event_type_keyboard()
    )

# Обробка вибору типу події
@dp.callback_query(lambda c: c.data.startswith("event_type:"))
async def type_state(callback: CallbackQuery, state: FSMContext):
    event_type, excludes = callback.data.split("&")
    event_type = event_type.split(":")[1]
    excludes = excludes.split(":")[1].split(";")

    await state.update_data(event_type=event_type) # Зберігаємо вибраний тип
події
    data = await state.get_data()
    try:
        # Запит подій конкретного типу
        events = await ticketmaster_api.fetch_all_events(data["country"],
data["city"], event_type)
    except ConnectionError:
        await callback.answer("Виникла помилка")
        return

    # Якщо подій за типом немає — пропонуємо обрати інший тип
    if events["page"]["totalElements"] == 0:
        await state.update_data(event_type="")
        excludes.append(event_type)
        await callback.message.edit_text("😣 За вашим типом подій, нічого не
знайдено. Оберіть інший",

reply_markup=event_type_keyboard(exclude=excludes))
        return

    # Зберігаємо отримані події у стані
    await state.update_data(events=events)
```

```
    # Відправляємо інформацію про кількість знайдених подій та клавіатуру з ними
    await callback.message.edit_text(
        f'У місті {data['city']} за категорією
{suported_event_types[event_type]} знайдено {events["page"]["totalElements"]}
подій',
        reply_markup=events_keyboard(events))

# Обробка навігації по сторінках списку подій
@dp.callback_query(lambda c: c.data.startswith("pag:"))
async def listing_events(callback: CallbackQuery, state: FSMContext, page=None):
    if page is None:
        typ = callback.data.split(":")[1]
        if typ == "page": # Кнопка з номером сторінки, нічого не робимо
            return
        page = callback.data.split(":")[-1]  # Отримуємо номер сторінки

    data = await state.get_data()
    # Запит подій на потрібній сторінці
    events = await ticketmaster_api.fetch_all_events(data["country"],
data["city"], data["event_type"], page=page)
    await state.update_data(events=events)
    await state.update_data(page=page)

    await answer_or_edit_with_or_without_photo(
        callback, None,
        f'У місті {data['city']} за категорією
{suported_event_types[data["event_type"]]} знайдено
{events["page"]["totalElements"]} подій',
        reply_markup=events_keyboard(events)
    )

# Обробка навігаційних кнопок ("На головну", "Інше місто", "Інший тип", "Назад"
тощо)
@dp.callback_query(lambda c: c.data.startswith("nav:"))
async def perform_navigation(callback: CallbackQuery, state: FSMContext):
    data = await state.get_data()
    where = callback.data.split(":")[-1]

    # Повернення до головного меню
    if where == "menu": # Повернення до головного меню
        await state.clear()
        return await start(callback.message, is_edit=True)

    # Введення іншого міста
    elif where == "search_city":
        await callback.message.edit_text("Введіть назву міста")
        await state.set_state(SearchState.city)
        return

    # Повернення до вибору типу події для поточного міста
    elif where == "event_type":
        await state.set_state(SearchState.city)
        return await city_state(callback.message, state,
city_param=data["city"])

    # Повернення до списку подій на поточній сторінці
    elif where == "event_list_page":
        page = (await state.get_data()).get("page", 0)

        return await listing_events(callback, state, page)


# Показ детальної інформації про конкретну подію
```

```python
@dp.callback_query(lambda c: c.data.startswith("event:"))
async def show_event(callback: CallbackQuery, state: FSMContext):
    ev_id = callback.data.split(":")[-1]
    data = await state.get_data()

    # Знаходимо подію в списку збережених подій
    event = [ev for ev in data["events"]["_embedded"]["events"] if ev["id"] ==
ev_id][0]

    name = event["name"]
    url = event.get("url", "")
    date = event.get("dates", {}).get("start", {}).get("localDate", "Невідомо")
    venue = event.get("_embedded", {}).get("venues", [{}])[0].get("name", None)
    price_info = event.get("priceRanges", [{}])[0]
    min_price = price_info.get("min", None)
    currency = price_info.get("currency", "")
    image = event.get("images", [{}])[0].get("url", "")

    # Формуємо текст повідомлення з інформацією про подію
    text = (
        f"🎵 <b>{name}</b>\n"
        + f"🗓 <b>Дата:</b> {date}\n"
        + (f"📍 <b>Місце:</b> {venue}\n" if venue else "")
        + (f"💰 <b>Вартість:</b> {min_price} {currency}\n" if min_price else "")
        + f"🔗 <a href='{url}'>Детальніше</a>"
    )
    # await callback.message.edit_media(image)
    await answer_or_edit_with_or_without_photo(callback, photo=image, text=text,
reply_markup=event_keyboard())


@dp.message()
async def echo(message: Message):
    await message.answer(message.text)


async def main():
    print("Bot satarted")
    await dp.start_polling(bot)


if __name__ == '__main__':
    asyncio.run(main())
```

ДОДАТОК Б – Файл config.py

```python
# Модуль для взаємодії з операційною системою, зокрема — для зчитування змінних середовища
import os

# Імпортуємо функцію для завантаження змінних середовища з .env-файлу
from dotenv import load_dotenv

# Завантажуємо змінні середовища з файлу `.env`, який розміщений у кореневій директорії проєкту
load_dotenv(dotenv_path='.env')

# Зчитуємо значення змінної BOT_TOKEN із середовища — токен Telegram-бота
BOT_TOKEN: str = os.getenv('BOT_TOKEN')
# Зчитуємо значення API-ключа Ticketmaster для запитів до їхнього API
TICKETMASTER_API_KEY: str = os.getenv('TICKETMASTER_API_KEY')
```

ДОДАТОК В – Файл keyboards.py

```
# Імпорт конструктора inline-клавіатур (з кнопками)
from aiogram.utils.keyboard import InlineKeyboardBuilder

# Список країн, які підтримуються для пошуку подій
suported_countries = [
    ("UK", "GB Велика Британія"),
    ("DE", "DE Німеччина"),
    ("ES", "ES Іспанія"),
    ("PL", "PL Польща"),
    ("HU", "HU Угорщина"),
    ("FR", "FR Франція"),
    ("DK", "DK Данія"),
]

# Словник із типами подій
suported_event_types = {
    "concerts": "🎵 Концерти",
    "festivals": "🎉 Фестивалі",
    "exhibitions": "🖼 Виставки",
    "sports": "🏅 Спорт",
    "all": "🌟 Всі події",
}

# Клавіатура з кнопкою для запуску пошуку
def search_keyboard():
    builder = InlineKeyboardBuilder()
    builder.button(text=f"Пошук події", callback_data="search")

    # Одна кнопка в рядку
    builder.adjust(1)
    return builder.as_markup() # Повертає готову розмітку клавіатури


# Клавіатура для вибору країни
def countries_keyboard():
    builder = InlineKeyboardBuilder()
    for code, name in suported_countries:
        # callback містить код країни
        builder.button(text=f"{name}", callback_data=f"country:{code}")

    # Кнопка повернення до меню
    builder.button(text=f"На головну", callback_data="nav:menu")
    # Всі кнопки по одній в рядку
    builder.adjust(1)
    return builder.as_markup()


# Клавіатура для вибору типу подій з виключенням деяких типів
def event_type_keyboard(exclude: list[str] = None):
    if exclude is None:
        exclude = []
    builder = InlineKeyboardBuilder()
    for code, name in suported_event_types.items():
        if code not in exclude:
            # callback містить тип події + перелік виключень, щоб не показувати
повторно
            builder.button(text=f"{name}",
callback_data=f"event_type:{code}&ex:{';'.join(exclude)}")
```

```
    builder.adjust(1)

    # Додаткові кнопки навігації
    nav = InlineKeyboardBuilder()
    nav.button(text="На головну", callback data="nav:menu")
    nav.button(text="Обрати інше місто", callback_data="nav:search_city")

    nav.adjust(2)
    # Додаємо кнопки навігації до основних
    builder.attach(nav)
    return builder.as_markup()


# Клавіатура зі списком подій + пагінація
def events_keyboard(events):
    builder = InlineKeyboardBuilder()
    for event in events["_embedded"]["events"]:
        builder.button(text=f"{event['name']}",
callback_data=f"event:{event['id']}")
    builder.adjust(1)

    # Кнопки пагінації
    pagination = InlineKeyboardBuilder()
    if events['page']['number'] != 0:
        pagination.button(text="<",
callback_data=f"pag:back:{events['page']['number'] - 1}")

    # Виводимо номер поточної сторінки / загальну кількість
    pagination.button(text=f"{events['page']['number'] +
1}/{events['page']['totalPages']}", callback_data="pag:page")

    if events['page']['number'] + 1 < events['page']['totalPages']:
        pagination.button(text=">",
callback_data=f"pag:next:{events['page']['number'] + 1}")
    pagination.adjust(3)

    # Кнопки навігації
    nav = InlineKeyboardBuilder()
    nav.button(text="На головну", callback_data="nav:menu")
    nav.button(text="Інше місто", callback_data="nav:search_city")
    nav.button(text="Інший тип", callback_data="nav:event_type")

    nav.adjust(3)

    # Приєднуємо пагінацію та навігацію до основної клавіатури
    builder.attach(pagination)
    builder.attach(nav)
    return builder.as_markup()

# Клавіатура перегляду події (одна кнопка — назад)
def event_keyboard():
    nav = InlineKeyboardBuilder()
    nav.button(text="Назад", callback_data="nav:event_list_page")

    return nav.as_markup()
```

ДОДАТОК Г – Файл loader.py

```
# Імпортуємо основні компоненти aiogram: Bot — для взаємодії з Telegram,
Dispatcher — для обробки апдейтів
from aiogram import Bot, Dispatcher
# Для задання параметрів за замовчуванням (наприклад, форматування)
from aiogram.client.default import DefaultBotProperties
# Форматування тексту повідомлень: HTML, Markdown тощо
from aiogram.enums import ParseMode
# Тимчасове збереження станів FSM у пам’яті
from aiogram.fsm.storage.memory import MemoryStorage

# Імпорт власного класу для роботи з Ticketmaster API
from tm_api import TICKETMASTER_API
# Імпорт токена Telegram бота та API-ключа Ticketmaster з конфігураційного файлу
from config import BOT_TOKEN, TICKETMASTER_API_KEY

# Створюємо екземпляр бота, передаючи йому токен та параметри за замовчуванням
(тут — форматування HTML)
bot = Bot(token=BOT_TOKEN,
default=DefaultBotProperties(parse_mode=ParseMode.HTML))
# Ініціалізуємо тимчасове сховище для FSM (Finite State Machine) — пам’ять.
storage = MemoryStorage()  # TODO connect redis

# Створюємо диспетчер, який буде обробляти всі події (апдейти), використовуючи
FSM-сховище
dp = Dispatcher(storage=storage)

# Ініціалізуємо клас-обгортку для роботи з Ticketmaster API, передаючи йому API-
ключ
ticketmaster_api = TICKETMASTER_API(TICKETMASTER_API_KEY)
```

ДОДАТОК Д – Файл states.py

```
# Імпортуємо базові класи для роботи з машиною станів (FSM) з бібліотеки aiogram
from aiogram.fsm.state import StatesGroup, State


# Визначаємо групу станів для сценарію пошуку подій
class SearchState(StatesGroup):
    # Стан, у якому користувач обирає країну (наприклад, "UK", "DE" тощо)
    country = State()

    # Стан, у якому користувач вводить або обирає місто
    city = State()

    # Стан, у якому користувач обирає тип події (наприклад, концерт, фестиваль
тощо)
    event_type = State()
```

ДОДАТОК Е – Файл tm_api.py

```python
# Імпортуємо aiohttp-сесію з aiogram для виконання HTTP-запитів у асинхронному режимі
from aiogram.client.session import aiohttp


# Клас для роботи з API Ticketmaster
class TICKETMASTER_API:
    # Ініціалізуємо клас токеном доступу до API
    def __init__(self, token):
        self.token = token # Зберігаємо API ключ для використання в запитах

    # Асинхронний метод для отримання списку подій
    async def fetch_all_events(self, country, city, event_type, size=10, page=0):
        # Отримати події з API Ticketmaster за параметрами країни, міста, типу події, розміру сторінки та номера сторінки.

        # Відкриваємо асинхронну HTTP-сесію
        async with aiohttp.ClientSession() as session:
            # Формуємо базовий URL для запиту до Ticketmaster API
            request_url = f"https://app.ticketmaster.com/discovery/v2/events.json?countryCode={country}&city={city}&size={size}&page={page}&apikey={self.token}"

            # Якщо обрано конкретний тип події, додаємо його до URL як параметр
            if event_type != 'all':
                request_url += f"&classificationName={event_type}"

            # Виконуємо GET-запит до сформованого URL
            response = await session.get(request_url)

            # Якщо запит успішний (код 200), повертаємо JSON-відповідь
            if response.status == 200:
                return await response.json()
            else:
                # Якщо сталася помилка — виводимо код помилки і відповідь, піднімаємо виключення
                print(response.status)
                print(await response.json())
                raise ConnectionError()
```

ДОДАТОК Є – Файл utils.py

```
# Імпортуємо необхідні типи з aiogram для роботи з повідомленнями
from aiogram import types


# Асинхронна універсальна функція для відповіді або редагування повідомлення, з фото або без
async def answer_or_edit_with_or_without_photo(
        message: types.Message | types.CallbackQuery, # Повідомлення або callback-запит
        photo: str | types.InputFile | None,    # Шлях до фото або об'єкт InputFile, або None
        text: str,      # Текст повідомлення
        reply_markup=None):     # Клавіатура (може бути None)

    # Якщо прийшов CallbackQuery, витягуємо оригінальне повідомлення
    if isinstance(message, types.CallbackQuery):
        message = message.message


    # Якщо є фото
    if photo:
        # Якщо вже є повідомлення з фото, і воно надіслане ботом — редагуємо лише медіа (фото і підпис)
        if message.content_type == "photo" and message.from_user.is_bot:
            await message.edit_media(types.InputMediaPhoto(media=photo, caption=text), reply_markup=reply_markup)
        else:
            if message.from_user.is_bot:
                    await message.delete()

            await message.answer_photo(photo, caption=text, reply_markup=reply_markup)

    # Якщо фото немає
    else:
        # Якщо бот надіслав старе повідомлення — видаляємо його
        if message.content_type == "photo" and message.from_user.is_bot:
            if message.from_user.is_bot:
                await message.delete()
            # Відправляємо нове текстове повідомлення
            await message.answer(text, reply_markup=reply_markup)
        else:
            # Якщо повідомлення надіслав бот — редагуємо його текст
            if message.from_user.is_bot:
                await message.edit_text(text, reply_markup=reply_markup)
            else:
                # Якщо повідомлення не від бота — просто надсилаємо нове повідомлення
                await message.answer(text, reply_markup=reply_markup)
```