ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Дослідження сучасного програмного інструментарію створення мобільних додатків для систем бронювання білетів.»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

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

_____ Денис ЩЕДРІН
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти
група КНДМ-61
Денис ЩЕДРІН

Керівник:
науковий ступінь, вчене звання
Сергій ПРОКОПОВ
к.т.н., доцент

Рецензент:
науковий ступінь, вчене звання
_____
(Ім'я, ПРІЗВИЩЕ)

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Щедріну Денису Олександровичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Дослідження сучасного програмного інструментарію створення мобільних додатків для систем бронювання білетів.

керівник кваліфікаційної роботи Сергій ПРОКОПОВ к.т.н., доцент,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, Бази даних.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Дослідження додатків для систем бронювання білетів

Аналіз технологій машинного навчання та можливостей застосування в мобільних додатках
Розробка бази даних

5. Перелік графічного матеріалу: *презентація*
    1. Розвиток мобільних додатків
    2. Характеристики мобільних додатків
    3. Kotlin
    4. Бази даних

6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконанн етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу розвитк технологій для мобільних додатків	05.11-12.11.23	
3	Дослідження баз даних	13.11-19.11.23	
4	Аналіз особливостей впливу штучного інтелекту н розробку мобільних додатків	20.11-25.11.23	
5	Дослідження технологій машинного навчання	27.11-03.12.23	
6	Застосування машинного навчання в мобільни додатках	04.12-10.12.23	
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
8	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти _____ Денис ЩЕДРІН
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Сергій ПРОКОПОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина магістерської роботи: 101 с., 2 рис., 16 джерел.

Об`єкт дослідження – сучасний програмний інструментарій для створення мобільних додатків.

Предмет дослідження – процес розробки мобільних додатків для систем бронювання білетів та сучасний інструментарій, який використовується у цьому процесі.

Наукове завдання – розробка мобільного додатку для бронювання білетів

Мета роботи – Метою дослідження є глибоке вивчення та аналіз сучасного програмного інструментарію для створення мобільних додатків у контексті систем бронювання білетів. Визначення та розгляд ключових технологій.

Методи дослідження – програмування, аналіз та порівняння існуючих методів, розробка додатка.

Провести літературний огляд для збору інформації про сучасні технології та інструменти у розробці мобільних додатків для бронювання білетів.

Розгляд та порівняння різних крос-платформених фреймворків (наприклад, React Native, Flutter, Xamarin) з точки зору їхньої придатності для розробки систем бронювання білетів.

Вивчення інструментів, які використовуються для створення та управління API систем бронювання, зокрема RESTful та GraphQL.

Створення прототипу мобільного додатка для бронювання білетів, використовуючи обрані технології та фреймворки.

Провести тестування розробленого прототипу для оцінки його ефективності, швидкодії та функціональності.

МОБІЛЬНІ ДОДАТКИ, ШТУЧНИЙ ІНТЕЛЕКТ, МАШИНЕ НАВЧАННЯ, БАЗИ ДАНИХ, KOTLIN, FIREBASE

ЗМІСТ

Вступ

У сучасному світі, де мобільні технології стають неодмінною частиною нашого повсякденного життя, розробка та вдосконалення мобільних додатків для бронювання білетів виявляється надзвичайно важливою задачею. З ростом популярності та доступності подорожей, а також культурних та розважальних подій, зростає і попит на ефективні та зручні засоби бронювання та керування білетами через мобільні пристрої.

У рамках дослідження буде проведений огляд існуючих мобільних додатків для бронювання білетів, розглянуті технології та тренди у розробці мобільних додатків, а також проведений аналіз програмного інструментарію, який використовується для їх створення. Це дослідження також включить практичний аспект - розробку мобільного додатка для бронювання білетів, що дозволить виявити переваги та недоліки використаних технологій та інструментів.

В результаті даного дослідження очікується отримання глибокого розуміння сучасних підходів до розробки мобільних додатків для бронювання білетів та надання рекомендацій для використання оптимальних інструментів у цій області. Зростання конкуренції на ринку мобільних додатків для бронювання білетів вимагає від розробників постійного вдосконалення функціоналу та ефективного використання сучасних технологій. Залучення клієнтів та створення зручного користувацького інтерфейсу, а також надійності та безпеки, стають ключовими аспектами успішної реалізації мобільного додатка.

Специфікація та вдосконалення систем бронювання білетів на

сьогоднішній день вимагає використання передових технологій та ефективного програмного інструментарію. У цьому контексті, обрання

правильного фреймворку та інструментів для розробки може визначити успіх проекту.

Однією з ключових відмінностей цього дослідження є практичний підхід, включаючи створення мобільного додатка для бронювання білетів. Цей практичний етап дозволить не лише теоретично розглянути питання, але й прагматично оцінити застосування обраних інструментів у реальному проекті.

Об’єкт дослідження – сучасний програмний інструментарій для створення мобільних додатків.

Предмет дослідження – процес розробки мобільних додатків для систем бронювання білетів та сучасний інструментарій, який використовується у цьому процесі.

Мета дослідження – Метою дослідження є глибоке вивчення та аналіз сучасного програмного інструментарію для створення мобільних додатків у контексті систем бронювання білетів. Визначення та розгляд ключових технологій.

Завдання дослідження:

1. Провести літературний огляд для збору інформації про сучасні технології та інструменти у розробці мобільних додатків для бронювання білетів.
2. Розгляд та порівняння різних крос-платформених фреймворків (наприклад, React Native, Flutter, Xamarin) з точки зору їхньої придатності для розробки систем бронювання білетів.
3. Вивчення інструментів, які використовуються для створення та управління API систем бронювання, зокрема RESTful та GraphQ
4. Створення прототипу мобільного додатка для бронювання білетів, використовуючи обрані технології та фреймворки.
5. Провести тестування розробленого прототипу для оцінки його ефективності, швидкодії та функціональност

1 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ СУЧАСНОГО ПРОГРАМНОГО ІНСТРУМЕНТАРІЮ МОБІЛЬНИХ ДОДАТКІВ

1.1 Еволюція технологій розробки мобільних додатків

Розробка мобільних додатків значною мірою еволюціонувала з моменту свого створення. Зручність доступу до багатьох програм безпосередньо з мобільного телефону полегшила наше життя, оскільки це дозволяє нам здійснювати платежі, робити покупки в Інтернеті, розважатися та отримувати інформацію з усього світу за кілька секунд. Окрім поширення смартфонів, високої швидкості Інтернету та розвитку платформ, у цю трансформацію є величезний внесок провідних компаній-розробників мобільних додатків.

1973 року компанія Motorola розробила перший мобільний апарат. Через 20 років на світ з'явився по суті перший кишеньковий комп'ютер — Psion 3 (рис. 1.1). Формально, вони існували і раніше (зокрема, того ж року з'явився перший

Рисунок 1.1 – Psion 3

смартфон IBM Simon), але обмежувалися лише записником, калькулятором та годинником.

Завдяки досвіду роботи з розробкою додатків для Android і iPhone, провідні розробники в усьому світі успішно радують користувачів інноваційними та багатофункціональними програмами, які приносять користь людям, компаніям і громадам.

Оскільки технології постійно розвиваються, наші мобільні програми стають потужнішими та розумнішими, інтегрованими з інноваційними функціями та надзвичайними можливостями.

А ось Psion 3 мав воістину безмежну функціональність: на базі операційної системи EPOC була створена мова програмування OPL (Open Programming Language), що дозволяє будь-кому бажаючому створити свою програму. Пізніше саме він ляже в основу відомої Symbian (рис 1.2).

Рисунок 1.2 – Symbian

У ті далекі часи у Psion був, по суті, лише один конкурент – Palm Pilot. Саме цей девайс не дозволив Psion 4 вийти на ринок. А в усьому справжній тачскрин і, головне, можливість створення додатків на C/C++. Неважко здогадатися, скільки переваг це дало платформі Palm OS.

Ось інструменти для роботи з мобільними гаджетами на той час:

WML (Wireless Markup Language) — мова розмітки, розроблена для пристроїв з обмеженнями, викликаними повільним інтернетом, малою кількістю оперативної пам'яті, розмірами та якістю екрану. По суті, це той же HTML, але з суворішими вимогами щодо парності тегів і сильно урізаною функціональністю. Вперше був застосований саме у Palm Pilot.

J2ME/JME, за аналогією з WML, є урізаною версією Java. Практично будь-яка людина, яка мала на початку 2000-х «серйозний» мобільний телефон, стикалася з пошуком програм, написаних на J2ME.

J2ME переважно складається з двох конфігурацій: CDC (Connected Device Configuration) та CLDC (Connected Limited Device Configuration). Перша призначена для відносно потужних та продуктивних пристроїв, наприклад, комунікаторів чи кишенькових комп'ютерів. Друга — для недорогих гаджетів із суттєвими обмеженнями, але із прицілом на широку аудиторію. Для мобільних телефонів використовувався режим MIDP (Mobile Information Device Profile), що забезпечує злагоджену роботу інтерфейсу, додатків, мережі та збереження всіх налаштувань. Саме завдяки йому ми пізнали дива мобільних 2D ігор та перших барвистих додатків.

Ключовий період розвитку мобільних додатків, що передбачає сучасний, практично повністю пов'язаний з існуванням системи Symbian. Symbian походить від платформи EPOC, в результаті спільної роботи Psion, Ericsson, Motorola та Nokia. Співпраця виявилася такою успішною, що до кінця епохи в 2009 році понад 250 мільйонів пристроїв працювало на Symbian.

Зрозуміло, основний внесок у розвиток системи внесла Nokia. Symbian S60 під їх керівництвом виріс у платформу, за потужністю та наповненістю порівнянну із сучасними. Незважаючи на те, що на S60 працювали смартфони Samsung і LG, домінування так і не переросло в однаковість.

Так, Sony Ericsson і Motorola були прихильниками Symbian UIQ, платформи з номінально ширшими можливостями, зокрема оптимізацією для роботи з сенсорними пристроями. Крім того, існували варіації Symbian S40, S80, S90 і всі вони були практично несумісні, що загострювало боротьбу між мобільними компаніями, а користувачі часто ставали прихильниками лише однієї марки телефонів.

При цьому сам фінський виробник на базі S60 періодично створював додаткові платформи для розробки ігор та програм, наприклад, N-Gage. Перша спроба у 2003 році провалилася, натомість друга у 2008 році ознаменувала вихід цілої лінійки мультимедійних пристроїв, що стали останньою передсенсорною класикою.

Явище світу Apple iPhone, а також небажання Nokia розлучатися зі спадщиною Symbian, визначило нову групу платформ, що протистоїть: iOS, Android, Windows Phone і BlackBerry OS.

Офіційно на світ система iOS (до 2010 року та запуску iPad носила назву iPhone OS) з'явилася лише в березні 2008 року, проте фактично існувала з початку 2007. Вся справа в тому, що Apple перші півтора роки дбайливо охороняли платформу від сторонніх розробників, стверджуючи, що смартфони працюють на урізаній десктопній операційній системі.

У 2008 році було випущено бета-версію середовища для розробки додатків - SDK (Software Development Kit). Сьогодні в пакет разом зі стандартними інструментами використання фізичних та програмних можливостей пристрою входить XCode та iPhone Simulator.

Розробка Android розпочалася у 2005 році: саме тоді Google купив молоду та амбітну компанію Android Inc. Достеменно невідомо, над якою саме платформою велася робота 2 роки, але через 10 місяців після старту продажів iPhone в Google

оголосили про запуск мобільної системи Android, створення Open Handset Alliance (OHL) - альянсу, що займається її підтримкою та розвитком, а також про пакет для розробників Android "Early Look" SDK.

Android заснований на ядрі Linux та віртуальній машині Java. Google зумів оперативно розробити і відразу дати всім бажаючим розробникам практично необмежені можливості для створення додатків - від Android Native Development Kit (портування бібліотек та компонентів з безлічі мов) до OpenGL ES (робота з тривимірною графікою). Крім того, з першого дня запуску в листопаді 2007 року, Google проводить конкурси на найкращі програми з багатомільйонними призами.

Windows Mobile жила і процвітала на ринку комунікаторів та кишенькових комп'ютерів починаючи з 2000 року, поки наприкінці десятиліття популярність сенсорних смартфонів із супутніми операційними системами не поставила Microsoft перед необхідністю створення гідної відповіді.

Ідея полягала в тому, щоб взяти від «старшого брата» все краще, прив'язати до телефонів Nokia, що все ще викликають довіру, і тим самим залучити розробників до освоєння нової платформи. Цій меті служила Visual Studio Express, що дозволяє створювати як вузькоорієнтовані програми, так і кросплатформні. Також WP пропонувала розробнику інструменти Windows Bridge (портування сторонніх програм), Expression Blend (веб-дизайн), XNA (ігри), Silverlight (інтернет-додатки). Втім, останні два фреймворки зараз уже не актуальні через відсутність підтримки та кросплатформної системи Windows 10.

BlackBerry OS - мабуть, найменш розвинена і найближча до вічного забуття мобільна операційна система. Перша версія була випущена в 2009 році, але лише через п'ять років вона стала по-справжньому стабільною і функціональною. Особливий акцент системи зроблений на зручність користування (багато цікавих

знахідок, на кшталт мініатюр, BB Hub, управління жестами були «впроваджені» пізніше в Android та iOS) та корпоративну безпеку.

А ось із додатками виявилося зовсім туго. Якщо сьогодні зайти на сторінку розробки, половиною запропонованих варіантів буде створення Android-програми з можливістю запуску на пристроях BB. Фактично ж, починаючи з 10 версії, ви отримуєте встановлений Amazon App Store і можливість вибрати будь-який інший Android-магазин.

Залишається тільки здогадуватися, що стане актуально завтра, як основна платформа, так і гаджета №1. З одного боку, очевидно, що рішення, актуальні сьогодні на «великих» пристроях будуть актуальними і на мобільних — веб-дизайн або створення додатків на Java.

Разом з тим, більш глибоке впровадження кишенькових девайсів (які тепер і на руці, і на обличчі) означає, що сенсорне керування завтра може змінитись, наприклад, повноцінно візуальною (мова жестів) або голосовою.

Не варто скидати з рахунків і можливе повернення до більш простих платформ мобільних телефонів. Зокрема, буквально днями стало відомо, що Microsoft розробила нову операційну систему для бюджетних апаратів, яка замінить сильно застарілі і Symbian, що все ще функціонують.

1.2 Крос-платформені фреймворки в сучасній мобільній розробці

З ростом ринку мобільних додатків та різноманіття платформ, розробники шукають ефективні способи створення додатків, які працюватимуть на різних платформах. Крос-платформені фреймворки виявилися важливим інструментом у

сучасній мобільній розробці. Проведемо невеликий огляд платформених фреймворків.

Xamarin. Xamarin - це фреймворк для кроссплатформенної розробки мобільних додатків (iOS, Android, Windows Phone) з використанням мови C #. Ідея дуже проста. Ви пишете код своєю улюбленою мовою, із застосуванням всіх звичних для вас мовних фіч нібито LINQ, лямбда-виразів, Generic`ів та async`ів. При цьому ви маєте повний доступ до всіх можливостей SDK платформи та рідного механізму створення UI, отримуючи на виході додаток, який, строго кажучи, нічим не відрізняється від нативних і (принаймні запевняє) не поступається їм у продуктивності.

Фреймворк складається з кількох основних частин:

- Xamarin.IOS – бібліотека класів для C#, що надає розробнику доступ до iOS SDK;
- Xamarin.Android - бібліотека класів для C #, що надає розробнику доступ до Android SDK;
- Компілятори для iOS та Android;
- IDE Xamarin Studio;
- Плагін для Visual Studio.

Певний час тому досить широку популярність здобули ряд фреймворків (наприклад PhoneGap), які пропонують розробку кросплатформових мобільних додатків на HTML5 з використанням JavaScript. Ідея полягає в тому, що програма розробляється як звичайний сайт для мобільних пристроїв з використанням відповідних js-бібліотек, наприклад, Jquery Mobile. Потім все це упаковується в контейнер, який для користувача виглядає як нативний додаток. Мінуси цих фреймворків очевидні: по-перше, ви не маєте доступу до нативних елементів UI. Тобто, навіть якщо ви хочете використовувати стандартну кнопку «Назад» для

iPhone, ви повинні її намалювати і зверстати. По-друге, ви отримуєте урізаний та узагальнений API для роботи з платформою. Таким чином, ті чи інші фічі, властиві якійсь окремій платформі, будуть вам недоступні. Ну і третє і найважливіше – така програма фізично запускається всередині браузера телефону (точніше всередині контролю WebView). Не потрібно розписувати довго, що це означає: низька продуктивність (особливо «хороший» WebView на старих версіях Android) і величезні проблеми з відображенням (ну, панове, це ж браузер). Хоча, звичайно, у певних випадках ці фреймворки можуть виявитися дуже доречними.

Xamarin заснований на open-source реалізації платформи .NET - Mono. Ця реалізація включає власний компілятор C#, середовище виконання, а також основні .NET бібліотеки. Мета проекту – дозволити запускати програми, написані на C#, на операційних системах, відмінних від Windows – Unix-системах, Mac OS та інших. Важливо, що розробкою Xamarin займаються самі люди, що й розробкою Mono. І (тут увага) - це НЕ Microsoft з усіма плюсами і мінусами.

З точки зору виконання додатків між iOS та Android є одна ключова відмінність - спосіб їх попередньої компіляції. Як відомо, для виконання програм в Android використовується віртуальна Java-машина Dalvik. Нативні програми, які пишуться на Java, компілюються в якийсь проміжний байт-код, який інтерпретується Dalvik`ом команди процесора в момент виконання програми (тобто аналогічно тому, як працює CLR в .NET). Це так звана Just-in-time компіляція (компіляція на льоту). В iOS використовується інша модель компіляції – Ahead-of-Time (компіляція перед виконанням). Xamarin враховує цю різницю, надаючи окремі компілятори для кожної з цих платформ, які дозволяють на виході отримувати справжні, нативні програми, які виконуються поза контекстом браузера і можуть використовувати всі апаратні та програмні ресурси платформи.

Для iOS ситуація проста – ніякої віртуальної машини немає і програмний код має бути просто заздалегідь скомпільований у машинний. Для цього використовується AOT компілятор Mono.

Для Android цікавіше. При компіляції програми відбувається переведення коду на C# в проміжний байт-код, зрозумілий віртуальній машині Mono і сама віртуальна машина також додається в упакований додаток. І Mono і Dalvik написані на «C» та працюють поверх ядра Linux (а ми пам'ятаємо, що Android заснована на Linux). Ви вже знаєте, що відбувається. При запуску програми на Android обидві віртуальні машини починають працювати пліч-о-пліч і обмінюються даними через спеціальний механізм wrapper`ів.

Бібліотека класів Monotouch.dll надає доступ до всіх можливостей iOS SDK. Для розробника це просто набір C#-класів з гарною анотацією. Всередині ці класи використовують розроблені інженерами Xamarin механізми біндингу на нативні класи та методи. Важливо, що цей механізм можна використовувати для биндинга будь-яких бібліотек, написаних на objective-c. Більшість класів і методів називаються так само, як в оригінальному iOS SDK, хоча бувають винятки (у цьому випадку доводиться використовувати пошук у документації Xamarin за оригінальною назвою, оскільки вона фігурує в атрибутах біндингу). У класах активно використовується механізм C# event`ів, що дозволяє писати гарний та компактний код обробників з використанням лямбда-виразів.

Для асинхронної розробки Xamarin надає можливість використовувати як класи з простору імен System.Threading.Thread і System.Threading.ThreadPool, так і повний спектр можливостей Task Parallel Library. Використання останньої, однак, вважається кращим. Крім того, на момент написання статті вийшла чергова версія Stable, в якій з'явилася підтримка .NET 4.5, зокрема, тепер можна використовувати ключові слова async/await. Хоча ця можливість була доступною і раніше, але для цього доводилося використовувати beta-канал оновлень.

Для кожної платформи Xamarin надає можливість використовувати нативні засоби розробки UI та нативні елементи інтерфейсу користувача. Для Android створення UI може відбуватися безпосередньо в коді або за допомогою декларативного підходу з описом інтерфейсу в XML. Для iOS це також або код, або використання нативних засобів проектування інтерфейсу - окремі xib-файли або один великий Storyboard. Редагування цих файлів відбувається у звичному для iOS-розробника середовищі XCode. І це означає, що вам знадобиться Mac. Так, для розробки iOS-додатків вам у будь-якому випадку знадобиться Mac з двох причин:

По-перше, як я вже сказав, для редагування UI у середовищі XCode. По-друге, для налагодження додатків потрібен симулятор iPhone/iPad, який доступний тільки на Mac.

Xamarin має відмінну документацію, що містить докладні посібники, сніпети, а також значну базу прикладів. Документація безпосередньо по всіх класах бібліотек Monotouch та Monodroid є частиною загальної документації Mono. Але, на жаль, цього все одно недостатньо, щоб покрити весь низку питань, які можуть виникати в процесі розробки. У Xamarin існує ком'юніті розробників, яке сконцентровано на офіційному форумі та на StackOverlow. Активністю та ініціативністю людей у ком'юніті похвалитися не можу. З п'яти запитань, поставлених на офіційному форумі, відповідь я отримав лише на одне. Можливо, чи то питав. У цьому плані неоціненну допомогу надала приватна тех. підтримка з інженерами електронною поштою, доступна в business-ліцензії. Відповідають, як правило, протягом кількох годин і не стандартними відписками «спробуйте вимкнути та включити», а дійсно розуміються на проблемі і допомагають її вирішити. Слід розуміти, що база питань та відповідей, накопичена для нативної розробки набагато ширша, ніж для Xamarin, тому, як би ви не хотіли, вам доведеться розібратися в специфічному синтаксисі objective-c (з Java проблем не повинно бути), щоб розуміти приклади коду на тому ж StackOverflow. Крім того,

це відкриє вам доступ до прочитання та розуміння офіційної документації для платформи, що на певному етапі може стати дуже важливим. З іншого боку, в цьому є й позитивний момент: отримавши такий базис, вам буде простіше перейти до нативної розробки за потреби.

Xamarin Studio - кросплатформова IDE, яка працює як на Mac OS X, так і на Windows. На вигляд ця вона виглядає дуже простою та доброзичливою, проте за зовнішній простий ховається досить потужний інструмент, який включив у себе безліч функцій, звичних нам у Visual Studio і Resharper:

Приємне підсвічування синтаксису;

- Автодоповнення коду (включаючи можливість одночасного імпорту namespaces);
- Зручний універсальний пошук за назвами файлів, типами, членами класів тощо;
- Розвиті можливості навігації за проектом: Швидкий перехід до опису класу, перехід до базового класу, перелік місць використання класу тощо;
- Різні механізми рефакторингу та швидка підказка (як alt+Enter у Resharper);
- Досить розвинені механізми дебага, включаючи стеження, перегляд поточного значення змінної при наведенні, візуалізацію потоків та аналог Immediate window VS;
- Вбудована інтеграція із системами контролю версій: SVN, Git та TFS (для TFS, щоправда, потрібні сторонні утиліти);

Xamarin пропонує можливість вести розробку в Visual Studio після встановлення спеціального плагіна, який доступний у business-ліцензії. Плюси очевидні: ви стаєте розробником мобільних програм, не змінюючи місця

дислокації, і можете використовувати всю важку артилерію в особі Resharper та інших ваших улюблених плагінів. Після встановлення плагіна для Visual Studio вам потрібно буде налаштувати з'єднання з Mac, яке буде використано при запуску проекту на виконання. Тобто. Після запуску, програма автоматично пересилається на Mac, де компілюється і завантажується або на симулятор або на пристрій, при цьому процес процес налагодження, розстановка брейкпоінтів і т.д. відбуватиметься у Visual Studio.

На даний момент технологія Xamarin є серйозним інструментом для вирішення складних завдань у галузі розробки мобільних додатків. Незважаючи на це, команда розробників не зупиняється і продовжує його активний розвиток та покращення. За останні два місяці помітні серйозні поліпшення загальної стабільності продукту. На мій погляд, у технології велике майбутнє і з кожним днем кількість розробників, які використовують її як основний фреймворк для розробки, буде неухильно зростати. Однак досить висока вартість ліцензії може стати перешкодою для використання на шляху indie-розробників. Загалом свій досвід роботи з даним фреймворком вважаю позитивним та продовжу його використання.

React Native. React Native - це крос-платформений фреймворк для розробки мобільних додатків. Він був розроблений компанією Facebook і випущений у 2015 році. Основна ідея React Native полягає в тому, щоб дозволити розробникам використовувати React, популярну бібліотеку для створення інтерфейсів користувача, для побудови мобільних додатків, які працюють на обох платформах: iOS та Android.

Крос-платформеність:

Одна з ключових переваг React Native - це можливість розробки додатків, які працюють як на iOS, так і на Android. Розробники можуть використовувати велику частину одного коду для обох платформ, що спрощує утримання та підтримку.

Використання JavaScript:

Розробка ведеться мовою JavaScript, яка є широко відомою та популярною серед веб-розробників. Це дозволяє використовувати та перевикористовувати навички, здобуті при створенні веб-застосунків.

React Native використовує декларативний підхід, де інтерфейс користувача описується за допомогою компонентів та їх взаємодії. Це спрощує розробку та робить код більш зрозумілим. React Native використовує нативні компоненти, тобто ті, які використовуються в розробці для кожної конкретної платформи. Це дозволяє досягти високої швидкодії та продуктивності, аналогічно до нативних додатків. React Native використовує схожий підхід до React, популярної бібліотеки для створення інтерфейсів користувача для веб-застосунків. Це дозволяє розробникам зручно переходити між веб- та мобільним розробкою. React Native підтримує живе перезавантаження, що дозволяє розробникам бачити зміни в коді в реальному часі без перезапуску додатка. Це прискорює процес розробки та полегшує виявлення помилок.

React Native знайшов широке використання в індустрії розробки мобільних додатків, і його спільнота розробників продовжує активно розвивати та підтримувати цей фреймворк. Багато великих компаній використовують React Native для розробки своїх мобільних додатків через його ефективність та продуктивність.

Хоча React Native є потужним інструментом для крос-платформеної мобільної розробки, він також має свої обмеження.

У деяких випадках, коли потрібно взаємодіяти зі складними або специфічними для платформи функціями, може знадобитися використання нативних модулів. Це може вимагати додаткового часу та зусиль для розробників. Хоча React Native забезпечує високу швидкодію, в порівнянні з повністю нативними додатками для окремих платформ може відзначатися невеликою затримкою. Для додатків, які вимагають максимальної продуктивності, може бути важливо вибрати повністю нативний шлях. Додатки, розроблені з використанням React Native, можуть мати більший розмір порівняно з нативними додатками. Це може впливати на швидкість завантаження та споживання пам'яті пристрою. У разі оновлення версії React Native може виникнути затримка в оновленні вже існуючих додатків. Розробники можуть стикатися із сумісністю різних версій та модулів. В деяких випадках може виникнути виклик із реалізацією складних елементів UI, які можуть бути легко втілені за допомогою нативних інструментів. Іноді в React Native може бути обмежений вибір бібліотек і плагінів порівняно з іншими крос-платформеними фреймворками або нативними рішеннями.

Не зважаючи на ці обмеження, React Native залишається одним з найпопулярніших та ефективних інструментів для крос-платформеної мобільної розробки, забезпечуючи гнучкість та продуктивність для багатьох розробників.

React Native має розвинену документацію та активну спільноту, яка надає важливі ресурси та підтримку для розробників.

Офіційна документація React Native - це основне джерело інформації, яке містить все необхідне для початку роботи з React Native. Вона включає в себе різні туторіали, гайди та API-документацію. GitHub-репозитарій React Native - тут ви знайдете вихідний код React Native, а також інші корисні ресурси, такі як інструкції щодо внесення внесків (contributing) та багато іншого.

Reactiflux - це чат для обговорення React та React Native. Це відмінне місце для отримання допомоги, дізнання новин та обміну досвідом. Stack Overflow - популярний ресурс для питань і відповідей, пов'язаних з React Native. Багато досвідчених розробників регулярно надають допомогу тут. React Native Blog - офіційний блог React Native, де публікуються новини, оновлення та цікаві статті.

Документація Expo - Expo є надбудовою над React Native, яка спрощує розробку та випробування додатків. Їх документація також є важливим ресурсом для розробників React Native.

У висновках можна висловити, що React Native виявився успішним та популярним фреймворком для крос-платформеної мобільної розробки. Важливими перевагами є крос-платформеність, реюзабельність коду, продуктивність та швидкість розробки, а також нативна швидкість та ефективність завдяки використанню нативних компонентів. Активна спільнота розробників та підтримка забезпечують стабільність та постійний розвиток фреймворку. Невеликі обмеження враховуються в контексті його переваг, і React Native продовжує залишатися високою вимогою для розробників та компаній, які прагнуть ефективно створювати мобільні додатки.

Більш того, вбудовані можливості React Native, такі як живе перезавантаження та гнучкість вибору між написанням коду для iOS та Android, роблять його привабливим вибором для розробників. Крім того, фреймворк використовує популярний та зручний мовний стек, що сприяє зниженню порогу входження для новачків.

Важливим елементом React Native є також його інтеграція з екосистемою React та можливість використання сторонніх бібліотек для розширення функціональності. Це робить його ідеальним вибором для широкого спектру проектів, включаючи не тільки стартапи, а й великі корпоративні застосунки.

З урахуванням наведених переваг та враховуючи індустрійні стандарти, React Native залишається важливим інструментом для мобільної розробки, допомагаючи розробникам швидко, ефективно та з успіхом створювати мобільні додатки для різних потреб.

Flutter. Flutter - молода, але дуже перспективна платформа, що вже привернула до себе увагу великих компаній, які запустили свої додатки. Цікава ця платформа своєю простотою, порівнянною з розробкою веб-додатків, і швидкістю роботи на рівні з нативними додатками.

Flutter, і React Native є міцними кросплатформними фреймворками для інтерфейсної розробки. Обидва чудово підходять для мобільних додатків. Обидва дадуть вам доступ до програм для iOS та Android. Обидва пропонують надійну продуктивність програми та сучасні рішення.

Flutter — це набір інструментів інтерфейсу користувача (користувальницького інтерфейсу) з відкритим кодом, розроблений Google для створення власно скомпільованих мобільних, веб-програм і програм для настільних комп’ютерів з єдиної кодової бази. Це дозволяє створювати красиві та високопродуктивні цифрові продукти з багатими настроюваними віджетами.

Flutter походить від Google і служить базовою технологією для багатьох відомих продуктів Google, таких як Google Pay і Google Ads.

Flutter був офіційно запущений як проект з відкритим вихідним кодом у травні 2017 року, що робить його доступним для ширшої спільноти розробників. У грудні 2018 року Flutter вийшов у свій перший стабільний випуск.

Фреймворк спочатку був націлений на мобільні платформи, зокрема пристрої iOS та Android. Він був створений як альтернатива існуючим кросплатформним структурам розробки. Згодом Flutter став популярнішим, ніж його основний конкурент React Native.

Оскільки Flutter продовжує набирати популярність, його екосистема процвітає. Велика кількість пакетів, бібліотек і інструментів продовжує розширюватися, надаючи розробникам багатий пул ресурсів. Це дозволяє швидше розробляти додатки, використовуючи існуючі рішення.

Flutter підтримує активну та співпрацюючу спільноту, де розробники можуть вільно ділитися знаннями, сприяючи її постійному зростанню та вдосконаленню.

Багато найкращих програм створено за допомогою Flutter, зокрема eBay Motors, Reflectly, Alibaba, Hamilton, Realtor.com, Square, Topline, Hookle, Birch Finance, Nubank, ING Bank Śląski (Польща) та Credit Agricole Bank Polska (Польща).

Flutter не є мовою програмування. Натомість це SDK для мобільних додатків (набір для розробки програмного забезпечення) для кросплатформної розробки додатків. Його також часто називають фреймворком Flutter.

SDK — це як готовий набір для створення програм. Це один пакет, який ви встановлюєте, і він містить усі необхідні інструменти, як-от інструмент для перетворення коду в програму, пошук помилок, а іноді навіть набір готових частин. Він створений для роботи з окремими пристроями та системами. Розробники використовують SDK, щоб легко додавати такі функції додатка, як оголошення та сповіщення.

Flutter SDK є повноцінним, надає повний набір інструментів для створення програм. У нього є власний механізм малювання для графіки, готові частини, які називаються віджетами, які можна додати у вашу програму, а також інструменти для перевірки того, як усе працює разом.

Рівень вбудовування використовує специфічну для платформи мову (наприклад, Java або Objective-C), щоб програми Flutter могли бездоганно працювати на різних операційних системах. Він діє як міст між нативним

машинним кодом, платформою та механізмом Flutter, забезпечуючи сумісність і плавне виконання.

Серцем Flutter є двигун Flutter, який в основному написаний мовою C++. Це як команда за лаштунками театру, яка займається підготовкою кожного додатка Flutter. Кожного разу, коли щось змінюється, на екрані з'являються всі візуальні елементи вашої програми. Це також потужний центр, який запускає всі основні функції Flutter, як-от створення графіки, упорядкування тексту, керування файлами та мережевою активністю, підтримка спеціальних можливостей, дозвіл доповнень, а також запуск і компіляція Dart, мови, на якій написані програми Flutter.

У Flutter термін «фреймворк» зазвичай стосується набору інструментів, бібліотек і функцій, які Flutter надає розробникам. Це включає в себе такі речі, як #widgets, які є готовими елементами, які можна використовувати та комбінувати для створення програми. Саме на цьому рівні відбувається магія реактивного програмування. наша програма може легко оновлюватися у відповідь на зміни, наприклад, з'являється повідомлення чату.

Ще більше розширюють структуру бібліотеки Material і Cupertino. Думайте про них як про тематичні набори інструментів: Material обслуговує дизайн у стилі Google, тоді як Cupertino адаптований до естетики Apple. Вони дозволяють використовувати базові компоненти рівня віджетів для створення програми, яка ідеально вписується в середовище Android або iOS.

По суті, фреймворк Flutter розроблений для оптимізації процесу створення користувацьких інтерфейсів, які динамічно реагують на події, такі як взаємодії користувача, і зберігають єдиний вигляд на різних платформах.

Також коротко про інші фреймворки які можуть використовуватись, такі як PhoneGap, Cordova та NativeScript.

PhoneGap. PhoneGap є комерційним продуктом Adobe, але базується на відкритому проекті Apache Cordova. Використовуючи PhoneGap, розробники можуть створювати мобільні додатки за допомогою веб-технологій, таких як HTML, CSS та JavaScript. Особливості PhoneGap:

- Простота використання та навчання.
- Крос-платформеність (підтримка iOS, Android, Windows Phone та інших).
- Використання веб-технологій для розробки.

Cordova (Apache Cordova). Apache Cordova — це відкритий фреймворк для крос-платформеної мобільної розробки. Він дозволяє використовувати стандартні веб-технології для створення мобільних додатків, аналогічно PhoneGap. Особливості Cordova:

- Крос-платформеність та можливість використання стандартних веб-технологій.
- Підтримка плагінів для розширення функціональності.
- Низький поріг входження для веб-розробників.

NativeScript. NativeScript є фреймворком для крос-платформеної мобільної розробки, який дозволяє використовувати JavaScript або TypeScript для створення нативних мобільних додатків. Особливості NativeScript:

- Нативна швидкість та вигляд для iOS та Android.
- Використання Angular або Vue.js для розробки інтерфейсу користувача.
- Доступ до нативних API без використання плагінів.

Вибір між цими фреймворками може залежати від ваших вимог, рівня експертизи, архітектурних вимог та інших факторів. Кожен з них має свої переваги та недоліки, і важливо розглядати їх у контексті конкретного проекту та ваших умов.

Переваги крос-платформених фреймворків:

Одноразовий код. Розробники можуть використовувати один і той самий код для створення додатків для різних платформ, що зменшує час та зусилля, необхідні для розробки та підтримки додатків.

Зниження витрат. Використання крос-платформених фреймворків може зменшити витрати на розробку, оновлення та підтримку додатків, оскільки розробники можуть працювати з однією кодовою базою.

Швидке розгортання на різних платформах. Додатки можуть бути швидко розгорнуті на різних платформах, оскільки код може бути перенесений з однієї платформи на іншу.

Більш легке управління командою. Одна команда розробників може працювати над розробкою для різних платформ, що полегшує управління та координацію проектом.

Широкий досяг ринку. Крос-платформені додатки можуть швидше дістатися до ринку, оскільки їх можна розгортати на різних платформах без значних модифікацій.

Спільні бібліотеки та компоненти. Можливість використовувати спільні бібліотеки та компоненти для різних платформ дозволяє прискорити розробку та покращити її якість.

Недоліки крос-платформених фреймворків:

Обмежена функціональність та витрати продуктивності. Крос-платформені додатки можуть мати обмежену функціональність або не забезпечувати такий високий рівень продуктивності, як нативні додатки.

Залежність від постачальників фреймворків. Розробники можуть бути обмежені функціональністю та оновленнями, які надає постачальник крос-платформеного фреймворку.

Проблеми з адаптацією до нативного вигляду. На деяких платформах можуть виникати труднощі з адаптацією до нативного вигляду, що може вплинути на користувацький досвід.

Збільшений розмір файлів та використання ресурсів. Крос-платформені додатки можуть вимагати додаткових ресурсів та мати більший розмір файлів порівняно з нативними додатками.

Проблеми зі швидкістю та ефективністю. Деякі крос-платформені фреймворки можуть мати проблеми зі швидкістю та ефективністю, особливо при роботі з важкими або великими додатками.

Наявність обмежень для визначених операційних систем. Деякі функції або можливості можуть бути обмежені або непідтримуваними на певних операційних системах.

Специфічні труднощі при оновленнях. Оновлення можуть вимагати додаткових зусиль, оскільки розробники повинні бути впевнені, що нова версія фреймворку сумісна з усіма платформами, на яких працює додаток.

Обираючи між крос-платформеним та нативним розробкою, розробники повинні ураховувати конкретні потреби та вимоги свого проекту, а також бізнес-цілі та обмеження ресурсів.

Зараз спостерігається зростання інтересу до крос-платформених рішень. Тенденції включають покращення продуктивності, набуття підтримки від великих компаній, які розробляють фреймворки, а також розвиток нових інструментів для полегшення розробки.

1.3 Роль та тенденції використання API в програмному інструментарії мобільних додатків

API відіграє ключову роль у взаємодії між мобільними додатками та серверами. Від традиційних RESTful підходів до більш гнучких технологій, таких як GraphQL, розробники шукають оптимальні способи забезпечення ефективного обміну даними та взаємодії з серверами бронювання. Основні аспекти ролі API включають в себе: Взаємодія з серверами, Інтеграція з іншими сервісами, Оновлення та синхронізація даних, Використання сторонніх сервісів та компонентів, Аналітика та відстеження.

Взаємодія з серверами за допомогою API в мобільних додатках є критично важливою для обміну даними між клієнтською частиною додатка та віддаленим сервером. Додатки можуть взаємодіяти з серверами для отримання різних видів даних, таких як тексти, зображення, відео, аудіо, та інше. Це може включати такі операції, як запити на отримання новин, профілю користувача, або інших важливих даних.

Користувачі можуть надсилати дані на сервер, наприклад, коли вони заповнюють форму, виконують покупку, надсилають повідомлення тощо. Взаємодія ця може використовувати HTTP-запити для відправлення даних на сервер. API використовуються для проведення процесу аутентифікації, щоб забезпечити ідентифікацію користувача. Одноразові токени, сесії та інші методи дозволяють забезпечити безпеку та контроль доступу до ресурсів. Взаємодія з серверами може включати в себе підтримку реального часу для отримання оновлень у режимі реального часу. Це може бути важливим для чатів, стрімінгу даних, оновлення стану.

Під час взаємодії можуть виникати помилки. Додатки повинні включати обробку помилок для адекватної відповіді на такі ситуації, наприклад, некоректний запит, відсутність з'єднання. Деякі дані можуть бути кешовані на мобільному пристрої для зменшення затрат на мережевий трафік та поліпшення швидкодії. Локальні сховища використовуються для тимчасового зберігання даних на пристрої. Зазвичай, взаємодія з серверами використовує HTTP-протокол для передачі даних. Однак деякі додатки можуть використовувати інші протоколи, такі як WebSocket для реального часу. Додатки можуть використовувати API для синхронізації даних між різними пристроями або відновлення даних після тимчасового втрати з'єднання.

Взаємодія з серверами через API є невід'ємною частиною розробки мобільних додатків і вимагає уважної реалізації та оптимізації для забезпечення ефективності, безпеки та хорошої користувацької досвіду.

Інтеграція з іншими сервісами в мобільних додатках є ключовим аспектом, який розширює функціональність та покращує користувацький досвід. Інтеграція з соціальними мережами дозволяє користувачам увійти до додатка, використовуючи свої облікові записи Facebook, Google, Twitter тощо. Крім того, додатки можуть підтримувати публікації на стіні чи обмін даними з соціальними платформами. Інтеграція з платіжними системами, такими як PayPal, Stripe або Apple Pay, дозволяє користувачам здійснювати покупки в додатку безпечно та ефективно. Це особливо важливо для е-комерційних додатків.

Використання картографічних сервісів, таких як Google Maps або Mapbox, дозволяє вбудовувати карти та навігацію в додаток. Це корисно для подачі географічної інформації та маршрутизації. Інтеграція з хмарними сховищами, такими як Google Drive, Dropbox або iCloud, дозволяє користувачам зберігати та синхронізувати дані між різними пристроями. Інтеграція з календарями, такими як

Google Calendar або Apple Calendar, може допомагати додатку взаємодіяти з подіями та завданнями користувача.

Використання медіа-сервісів, таких як YouTube або Vimeo, дозволяє вбудовувати відео в додаток. Також може використовуватися інтеграція з музичними сервісами, такими як Spotify або Apple Music. Інтеграція з електронною поштою та сервісами сповіщень дозволяє додаткам взаємодіяти з користувачами через електронну пошту або сповіщення на пристрої. Деякі додатки інтегруються з IoT-пристроями та смарт-технологіями, дозволяючи користувачам керувати своїм домом, автомобілем чи іншими підключеними пристроями.

Інтеграція з іншими сервісами відкриває нові можливості для функціональності додатків та створює більш зручний та збалансований користувацький досвід. При цьому важливо дотримуватися стандартів безпеки та захисту приватності при обробці зовнішніх сервісів.

Оновлення та синхронізація даних в мобільних додатках грають ключову роль у забезпеченні актуальності та доступності інформації для користувачів.

Для забезпечення актуальності даних додатки можуть використовувати автоматичні оновлення. Це може бути важливо для додатків новин, метео, фінансових сервісів тощо, де актуальність інформації є критичною. Деякі додатки дозволяють користувачам вручну оновлювати дані. Наприклад, у соціальних мережах користувач може оновлювати стрічку або у додатках для здоров'я - оновлювати дані про кроки чи пульс. Для користувачів, які використовують додаток на різних пристроях (наприклад, смартфоні та планшеті), синхронізація дозволяє узгоджувати дані між цими пристроями. Це може включати налаштування, взаємодію з завданнями або збереження вибраного контенту.

Для покращення користувацького досвіду у ситуаціях з обмеженим або відсутнім інтернет-з'єднанням додатки можуть забезпечувати можливість роботи в

офлайн-режимі, а пізніше синхронізувати дані, коли з'єднання стає доступним. Деякі додатки використовують кешування для збереження копій даних на пристрої користувача. Це дозволяє прискорити завантаження даних та зменшити навантаження на сервер. У випадках синхронізації даних між різними пристроями чи користувачами важливо контролювати конфлікти даних. Додатки повинні вміло вирішувати ситуації, коли дані змінюються на кількох пристроях одночасно. Під час синхронізації та оновлення даних важливо дотримуватися стандартів безпеки. Використання шифрування, токенів авторизації та інших методів захисту допомагає убезпечити дані користувачів від несанкціонованого доступу. Для деяких додатків важливо регулярно оновлювати дані віддалено або розповсюджувати повідомлення про нові дані. Це може бути важливим для додатків, які працюють з розкладами, новинами або іншою часово-залежною інформацією.

Загально кажучи, оновлення та синхронізація даних є ключовими аспектами для забезпечення ефективної роботи мобільних додатків та задоволення потреб користувачів у швидкому та актуальному доступі до інформації.

Використання сторонніх сервісів та компонентів є важливою стратегією в розробці мобільних додатків, оскільки це дозволяє розробникам використовувати готові рішення та функціональність, що спрощує процес розробки та покращує якість додатку. Використання готових UI-компонентів (бібліотеки або фреймворки) дозволяє швидко створювати інтерфейси з високою якістю. Наприклад, Material Design для Android або UIKit для iOS. Крім того, бібліотеки, такі як React Native Elements або Flutter Widgets, надають готові компоненти для крос-платформеного використання. Інтеграція з API соціальних мереж дозволяє користувачам увійти або реєструватися через свої облікові записи Facebook, Google, Twitter тощо. Також можуть використовуватися бібліотеки для авторизації, такі як Firebase Authentication. Використання готових платіжних систем, таких як

Stripe або PayPal, дозволяє додатку проводити безпечні та зручні операції оплати за товари чи послуги.

Сервіси аналітики, такі як Google Analytics або Flurry, дозволяють збирати та аналізувати дані про використання додатку. Це може допомогти в оптимізації та вдосконаленні користувацького досвіду. Використання хмарних сервісів, таких як Firebase, AWS або Microsoft Azure, дозволяє легко зберігати та синхронізувати дані між різними пристроями. Це також може включати збереження файлів, баз даних та інші хмарні рішення. Використання сервісів аналізу зображень, таких як Google Vision або Microsoft Azure Computer Vision, дозволяє додаткам впроваджувати функціональність розпізнавання обличчя, аналізу об'єктів на зображеннях. Додатки можуть використовувати сервіси для реального часу, такі як Firebase Realtime Database або WebSocket, для побудови функціональності чату, спільної роботи та оновлення в реальному часі. Використання геолокаційних сервісів, таких як Google Maps або Mapbox, дозволяє додаткам надавати функціональність карт та навігації.

Використання сторонніх сервісів та компонентів дозволяє розробникам значно прискорити розробку додатків, зосереджуючись на унікальних аспектах свого продукту та полегшуючи інтеграцію з популярними технологіями та сервісами.

Аналітика та відстеження в мобільних додатках грають важливу роль у зборі та аналізі даних про використання додатку. Це дозволяє розробникам отримувати інсайти щодо поведінки користувачів, виявляти проблеми та покращувати користувацький досвід. Збір загальних даних про використання додатка, таких як кількість відвідувань, активність користувачів, час, проведений у додатку. Відстеження взаємодії користувачів з інтерфейсом додатка, наприклад, визначення, які екрани викликають найбільший інтерес, або які функції використовуються частіше за інші. Вимірювання швидкості завантаження додатка,

часу відгуку та інших параметрів продуктивності для забезпечення оптимальної роботи додатка.

Google Analytics та Firebase Analytics. Дозволяють відстежувати різні події та дії користувачів у додатку, отримувати дані про конверсію та використання.

Flurry. Надає інструменти для аналізу користувацької активності та трендів, дозволяє створювати власні звіти.

Mixpanel. Спеціалізується на аналізі продуктової аналітики та ретеншена користувачів.

Apple Analytics. Для додатків на iOS дозволяє отримувати інсайти через інструменти в Apple Developer.

Визначення ключових подій у додатку, таких як використання певної функції чи завершення певної дії. Аналіз конверсій дозволяє визначити ефективність різних етапів взаємодії з додатком. Використання A/B тестів для порівняння різних варіантів додатку та визначення, який варіант приводить до кращого взаємодії користувачів та досягнення поставлених цілей. Аналіз статистики збоїв та помилок дозволяє оперативно виявляти проблеми та швидко виправляти їх через випуск відповідних оновлень. Аналіз ретеншена користувачів допомагає визначити, наскільки довго користувачі залишаються активними в додатку. Прогнозування майбутнього використання дозволяє розробникам адаптувати стратегії. Важливо дотримуватися стандартів захисту особистих даних та приватності користувачів при зборі та аналізі даних. Використання аналітики та відстеження є невід'ємною частиною стратегії розробки мобільних додатків. Це дозволяє розробникам приймати обґрунтовані рішення, вдосконалювати додаток та надавати кращий користувацький досвід.

Тенденції використання API в програмному інструментарії мобільних додатків:

RESTful API (Representational State Transfer) - це стиль архітектури для розробки веб-сервісів, який базується на принципах та властивостях HTTP. Такий підхід спрощує взаємодію між різними компонентами системи, забезпечуючи легкість розуміння, масштабованість та надійність. У RESTful API вся інформація вважається ресурсами, які можуть бути представлені через URI (Uniform Resource Identifier). Наприклад, для об'єкта "користувач" URI може виглядати як /users.

REST використовує основні HTTP методи для взаємодії з ресурсами:

- GET: Отримання інформації про ресурс.
- POST: Створення нового ресурсу.
- PUT: Оновлення існуючого ресурсу або створення, якщо його не існує.
- DELETE: Видалення ресурсу.
- PATCH: Часткове оновлення ресурсу.

Ресурси можуть бути представлені у різних форматах, таких як JSON, XML або HTML, залежно від потреб клієнта та налаштувань сервера. Зазвичай, JSON є найпоширенішим форматом. Кожен запит до сервера повинен містити достатньо інформації для зрозуміння, як обробити цей запит. Сервер повинен бути станонезалежним, що означає, що вся інформація для обробки запиту повинна бути включена в запит. Кожен запит клієнта до сервера повинен містити всю необхідну інформацію для обробки запиту, і сервер не повинен зберігати жодного стану про клієнта між запитами. Для забезпечення зворотної сумісності та підтримки різних версій API рекомендується включати версію API в URI або заголовок запиту.

RESTful API може надавати можливості фільтрації та сортування даних. Це дозволяє клієнтам отримувати тільки необхідні дані та сортувати їх за різними параметрами. Використання стандартів безпеки, таких як HTTPS, для захисту передачі даних між клієнтом та сервером. Також може включати автентифікацію та авторизацію користувачів. Використання кешування для прискорення доступу

до ресурсів та зменшення навантаження на сервер. Надання докладної документації для розробників, яка описує доступні ресурси, їхні URI, параметри запитів та формати даних.

RESTful API надає простий та ефективний спосіб взаємодії між різними компонентами веб-системи. За допомогою стандартів HTTP та принципів REST, розробники можуть побудувати ефективні та легко розширювані інтерфейси для взаємодії з додатками.

GraphQL - це мова запитів та середовище виконання для взаємодії з API, яке було розроблено компанією Facebook. Основним відмінником GraphQL є те, що він дозволяє клієнтам зазначати, які саме дані вони хочуть отримати, а не отримувати повний набір даних, як це часто відбувається в REST API.

GraphQL описує ваші дані як граф, де кожен вузол у графі представляє об'єкт, а ребра визначають відносини між цими об'єктами. Це дозволяє клієнтам ефективно запитувати лише ті дані, які їм потрібні. Клієнт може вказати саме ті поля, які йому потрібні, і отримати дані у форматі, який йому зручний. Це дозволяє уникнути зайвої передачі непотрібних даних і зменшити кількість запитів. У GraphQL всі дані є об'єктами, і всі запити обробляються як граф об'єктів. Це робить взаємодію з API послідовною та передбачуваною. GraphQL використовує систему типів для визначення, які дані можна запитувати та як вони повинні повертатися. Це дозволяє забезпечити консистентність та безпеку запитів. Окрім запитів на отримання даних, GraphQL підтримує мутації, які дозволяють змінювати дані або виконувати дії на сервері, такі як додавання, оновлення чи видалення записів. GraphQL також підтримує підписки, які дозволяють клієнтам отримувати реальні часи або асинхронні оновлення від сервера при зміні даних. GraphQL надає зручні інструменти для автоматичної генерації документації API, включаючи всі доступні типи, запити та мутації. Одним з ключових переваг GraphQL є можливість виконувати один запит на отримання всієї необхідної інформації, уникнувши

ситуацій, коли клієнту необхідно робити кілька запитів на різні ендпоінти для отримання повної інформації.

GraphQL не замінює REST, але надає альтернативний підхід до взаємодії з API, особливо в ситуаціях, де потрібна гнучкість та ефективність при обміні даними між клієнтом та сервером.

Мобільні SDK (Software Development Kit) та бібліотеки - це інструментарій для розробників, який включає набір інструментів, документацію та кодові зразки для спрощення та прискорення процесу створення мобільних додатків.

Операційні Системи:

- iOS SDK (iOS Software Development Kit): Набір інструментів для розробки додатків для пристроїв, що працюють під управлінням iOS (iPhone, iPad, iPod Touch).
- Android SDK: Інструментарій для розробки додатків для платформи Android.

Хмарні Сервіси та Бази Даних:

- Firebase: Платформа від Google, яка надає набір сервісів, таких як база даних в реальному часі, аутентифікація, збереження файлів, аналітика та інше.
- AWS Mobile SDK: Набір інструментів для розробки мобільних додатків на платформі Amazon Web Services.

Крос-платформені Фреймворки:

- React Native: Фреймворк від Facebook, який дозволяє розробляти мобільні додатки за допомогою JavaScript та React.
- Flutter: Фреймворк від Google, що використовує Dart для створення красивих та високоефективних інтерфейсів.

Графіка та UI:

- UIKit (iOS): Фреймворк для розробки користувацького інтерфейсу для iOS додатків.
- Android Jetpack: Набір бібліотек та інструментів для розробки Android додатків, що включає компоненти для роботи з UI, навігації, базами даних та іншими аспектами розробки.

Мережеві Бібліотеки:

- Alamofire (iOS): Бібліотека для роботи з мережею на платформі iOS, написана на Swift.
- OkHttp (Android): Бібліотека для роботи з мережею на платформі Android, яка є частиною бібліотеки Android Jetpack.

Мобільні Аналітичні Інструменти:

- Google Analytics для мобільних додатків: Сервіс для відстеження та аналізу активності користувачів в мобільних додатках.
- Flurry Analytics: Платформа для аналізу даних про використання мобільних додатків.

Автентифікація та Безпека:

- OAuth (Open Authorization): Протокол для автентифікації користувачів через сторонні служби.
- Firebase Authentication: Сервіс для реалізації безпечної аутентифікації в мобільних додатках.

Інструменти для Тестування:

- XCTest (iOS): Фреймворк для автоматизованого тестування на платформі iOS.

- Espresso (Android): Фреймворк для автоматизованого тестування на платформі Android.

Асистенти та Інструменти Розробки:

- Xcode (iOS): Інтегроване середовище розробки для платформи iOS.
- Android Studio: Інтегроване середовище розробки для платформи Android.

Розпізнавання та Обробка Зображень:

- ML Kit (Firebase): Набір інструментів для машинного навчання, що дозволяє виконувати завдання розпізнавання зображень, перекладу тексту та інших завдань.

Ці інструменти та бібліотеки значно полегшують та прискорюють розробку мобільних додатків, роблячи її більш ефективною та менш витратною за часом. Розробники можуть використовувати ці інструменти для реалізації різноманітних функцій та оптимізації продуктивності своїх додатків.

Мікросервісна архітектура - це архітектурний підхід до розробки програмного забезпечення, при якому додаток розбивається на невеликі і незалежні мікросервіси, які можуть функціонувати та розвиватися незалежно один від одного. Кожен мікросервіс відповідає за конкретну бізнес-функцію та взаємодіє з іншими мікросервісами через мережу, зазвичай за допомогою API.

Кожен мікросервіс представляє собою самодостатню компоненту з власною базою даних та логікою бізнес-процесу. Це дозволяє розробникам працювати над окремими частинами системи незалежно одна від одної. Кожен мікросервіс може бути розгорнутий, оновлений та масштабований незалежно від інших мікросервісів. Це забезпечує велику автономію розробникам та командам. Мікросервіси взаємодіють між собою через визначені API, що забезпечує

стандартизований спосіб обміну даними та викликів. Здатність масштабувати кожен мікросервіс окремо дозволяє ефективно використовувати ресурси та забезпечити високу продуктивність. Мікросервіси можуть бути легко змінені та адаптовані до нових вимог. Це дозволяє швидко реагувати на зміни в бізнес-вимогах.

Мікросервісна архітектура сприяє використанню DevOps-практик та автоматизації розгортання та управління інфраструктурою. Кожен мікросервіс може використовувати свою власну базу даних, що полегшує управління даними та розвиток окремих компонент. Для відстеження та аналізу діяльності кожного мікросервісу. Кожен мікросервіс повинен мати власні засоби безпеки, такі як автентифікація та авторизація. Для управління роботою та координації мікросервісів.

Мікросервісна архітектура дозволяє компаніям створювати гнучкі та масштабовані додатки, але вона також вимагає уваги до деталей, таких як культура команди, автоматизація та вибір правильних технологій для кожного мікросервісу.

Висновки до розділу 1

У цьому розділі ми детально розглянули теоретичні аспекти розробки програмного інструментарію для мобільних додатків. Зазначили, що сучасні технології дозволяють розробникам вибирати серед різноманітних фреймворків, підходів та платформ для створення додатків, що відповідають конкретним бізнес-вимогам та викликам сучасного ринку.

Ми розглянули різні аспекти розробки, включаючи нативну та крос-платформену розробку, мікросервісну архітектуру, та важливі компоненти, такі як

мобільні SDK та бібліотеки. Аналізуючи переваги та недоліки кожного підходу, ми визначили, що вибір певного інструменту чи методології повинен залежати від конкретних вимог проекту, ресурсів, доступних для розробки, та стратегії бізнесу.

Також, розглянувши еволюцію технологій розробки мобільних додатків, ми визначили ключові тенденції, які визначають сучасний підхід до створення високопродуктивних та ефективних мобільних рішень.

У висновку першого розділу ми визначили, що розробка сучасного програмного інструментарію для мобільних додатків — це складний та динамічний процес, який вимагає обдуманого вибору технологій та підходів. При розгляді теоретичних аспектів було важливо врахувати велику різноманітність варіантів та нюансів, що можуть впливати на вибір конкретного інструменту чи стратегії розробки.

Ми висвітлили протистояння між нативною та крос-платформеною розробкою, підкресливши переваги та недоліки кожного підходу. Зрозуміло, що обираючи між ними, розробники повинні враховувати специфіку свого проекту, вимоги до продуктивності, ресурси та строк постачання.

Досліджуючи мікросервісну архітектуру, ми зазначили, що цей підхід може виявитися важливим для створення масштабованих та еластичних мобільних додатків, які легко можна адаптувати до змін в бізнес-вимогах.

Ми також визначили ключові компоненти мобільного програмного інструментарію, такі як мобільні SDK та бібліотеки, які грають важливу роль у розробці та оптимізації функціональності додатків.

Зазначимо, що в епоху стрімкої еволюції технологій, розробники мають у своєму розпорядженні різноманітні інструменти та підходи, які дозволяють їм ефективно створювати інноваційні та конкурентоспроможні мобільні рішення. У наступних розділах дипломної роботи буде проведено більш детальний аналіз

конкретних технологій та їх застосування для створення програмного інструментарію мобільних додатків з урахуванням врахування вивчених теоретичних засад.

Отже, цей розділ надав основоположний огляд теоретичних аспектів, які визначають ландшафт розробки мобільних додатків у сучасному інформаційному суспільстві. У наступних розділах буде проведено більш детальний аналіз конкретних технологій та підходів з метою створення обґрунтованого та ефективного програмного інструментарію для мобільних додатків.

2 СУЧАСНІ ПРОГРАМНІ ІНСТРУМЕНТАРІЇ

2.1 Інструменти розробки програмного забезпечення

Інтегроване середовище розробки (ICP) — це програмна платформа, яка надає розробникам інструменти для написання, тестування і налагодження програмного забезпечення. Ці інструменти зазвичай об'єднуються в єдиному інтерфейсі, що спрощує роботу розробників та забезпечує єдність робочого процесу. Давайте розглянемо основні аспекти інтегрованих середовищ розробки: текстовий редактор, компілятор та інтерпретатор, відкладчик, управління версіями, система підсвічування синтаксису та автодоповнення, інтеграція інструментів розробки, допоміжні інструменти.

Текстовий редактор – це програмний інструмент, який призначений для створення та редагування текстового вмісту. В контексті програмування та розробки програмного забезпечення, текстові редактори використовуються для написання та редагування вихідних кодів програм. Текстовий редактор дозволяє вам створювати, відкривати та редагувати текстові файли. Він може підтримувати різні формати текстових файлів, такі як TXT, HTML, XML, а також мови програмування, такі як Java, Python, C++, тощо. Функція підсвічування синтаксису надає кольорове виділення різних елементів мови програмування чи мови розмітки, що полегшує читання та розуміння коду.

Деякі текстові редактори мають функцію автодоповнення, яка автоматично завершує слова або конструкції коду, що полегшує написання програм та запобігає деяким помилкам. Текстові редактори можуть підкреслювати потенційні помилки в коді або вказувати на них іншим чином, щоб розробники могли легше виявити та виправити проблеми. Деякі текстові редактори надають додаткові функції, такі як

інтеграція з системами контролю версій, підтримка плагінів, розширена відладка тощо. Зручне керування вікнами та вкладками дозволяє легко перемикатися між різними файлами та проектами. Деякі приклади текстових редакторів включають Visual Studio Code, Sublime Text, Atom, Notepad++, Vim, Emacs та інші. Вибір текстового редактора залежить від індивідуальних вподобань та потреб розробника.

Компілятор та інтерпретатор. Компілятор – це програма, яка перетворює вихідний код програми, написаний на високорівневій мові програмування, в еквівалентний йому низькорівневий машинний код або код байткоду. Компілятор аналізує весь вихідний код програми і генерує еквівалентний йому виконуваний файл. Цей виконуваний файл можна запустити на виконання без необхідності наявності вихідного коду.

Інтерпретатор – це програма, яка виконує вихідний код програми, перекладаючи його в машинний код або інший виконуваний формат на льоту, без попереднього створення окремого виконуваного файлу. Інтерпретатор аналізує і виконує вихідний код по одній команді або рядку коду одразу, без попереднього компіляційного етапу. Це забезпечує більшу гнучкість, але може призвести до меншої продуктивності порівняно з компіляцією.

Основні відмінності:

- Час виконання: Компілятор вимагає часу на попередню компіляцію програми перед виконанням, тоді як інтерпретатор може виконувати код безпосередньо під час його введення.
- Виконуваний файл: Компілятор генерує виконуваний файл або бібліотеку, тоді як інтерпретатор виконує вихідний код без створення окремого виконуваного файлу.

- Швидкість виконання: Зазвичай компільований код працює швидше, оскільки він не потребує перекладу під час виконання. Проте інтерпретований код може забезпечити більшу гнучкість.
- Переносимість: Виконуваний код компілятора специфічний для платформи, тоді як інтерпретований код може бути переносимим, оскільки він залежить від інтерпретатора.

Обидва підходи мають свої переваги та недоліки, і вибір між компіляцією та інтерпретацією часто залежить від вимог конкретного проекту та вибору мови програмування.

Відладчик (англ. debugger) – це програмний інструмент, призначений для виявлення та усунення помилок (багів) у вихідному коді програми. Відладка є важливою частиною процесу розробки програмного забезпечення і дозволяє розробникам ефективно виправляти помилки та забезпечувати правильну роботу програм.

Розробник може встановлювати точки зупинки в коді, де виконання програми призупиниться, щоб розробник міг аналізувати стан програми на цій точці. Відладчик дозволяє розробникам виконувати програму по одному рядку коду або функції за раз, щоб слідкувати за змінами стану програми. Розробник може переглядати значення змінних та вивід на екран у будь-який момент виконання програми. Відладчик відображає стек викликів, тобто ієрархію викликів функцій, що дозволяє розробнику розуміти порядок виконання програми.

Деякі відладчики дозволяють розробникам відстежувати використання пам'яті, щоб виявити можливі проблеми з витіканням пам'яті чи дефектами. Відладчик може автоматично призупиняти виконання програми у випадку виняткових ситуацій (exceptions) для аналізу та виправлення проблем. Деякі відладчики надають можливість відлагодження віддалено, що корисно для

відлагодження програм на віддалених серверах чи в середовищах віртуалізації. Деякі відладчики можуть надавати інформацію про продуктивність коду, допомагаючи розробникам знаходити та оптимізувати швидкодіючі елементи програми. Ефективне використання відладчика важливо для швидкого та якісного вирішення проблем у програмному коді.

Управління версіями (Version Control) – це система, яка дозволяє розробникам відстежувати та керувати змінами в коді та ресурсах проекту протягом часу. Вона дозволяє зберігати історію змін, відстежувати версії коду та спільно працювати над проектом. Основні концепції та функції систем управління версіями:

- Репозиторій є централізованим сховищем, де зберігається весь код та інші ресурси проекту, а також історія змін.
- Коміт - це дія, яка фіксує конкретні зміни в коді та інших файлах та додає їх до історії проекту в репозиторії.
- Гілка - це окрема лінія розвитку, яка може включати у себе новий функціонал чи зміни безпеки. Робота в гілках дозволяє паралельно розробляти різні функції чи виправляти помилки.
- Злиття - процес об'єднання змін з однієї гілки в іншу. Злиття може бути автоматизованим або викликати конфлікти, які потрібно вирішити вручну.
- Відгалуження - це створення копії репозиторію, яка може існувати незалежно. Відгалуження дозволяє працювати над великими змінами або експериментувати без впливу на основний код.
- Запит на злиття - це механізм, що дозволяє розробникам пропонувати свої зміни для обговорення та включення в основну гілку проекту.
- СУВ веде історію змін, що дозволяє переглядати, хто і коли вніс зміни, а також що саме було змінено.

- Конфлікти виникають, коли один і той самий рядок коду змінюється в різних гілках. Розробники повинні розробити стратегію об'єднання конфліктуючих змін.

Популярні системи управління версіями включають Git, Mercurial, Subversion та інші. Вони є невід'ємною частиною сучасного розробницького процесу, дозволяючи ефективно та забезпечуючи колективну роботу над програмним забезпеченням.

Система підсвічування синтаксису. Система підсвічування синтаксису це функція текстового редактора чи інтегрованого середовища розробки (ICP), яка візуально виділяє різні елементи програмного коду за допомогою кольорів чи інших візуальних ефектів. Це полегшує сприйняття та редагування коду, оскільки визначення синтаксичних елементів стає візуально очевидним. Переваги системи підсвічування ситаксису, вона забезпечує кращу читабельність коду, виявляє помилки та виділяє ключові конструкції та покращує загальний зовнішній вигляд коду, роблячи його біль структурованим та зрозумілим.

Автодоповнення – це функція, яка автоматично доповнює код під час його введення. Вона може пропонувати доступні функції, методи, змінні та інші елементи, що спрощує процес написання коду та зменшує кількість помилок. Серед переваг, можна виділити: збільшення швидкості написання коду, зменшення кількості помилок через автоматичне визначення правильних ідентифікаторів, допомагає розробникам вивчати API та бібліотеки, виводячи доступні функції та їхні параметри.

Системи підсвічування синтаксису та автодоповнення є важливими елементами комфортного та продуктивного робочого процесу для розробників програмного забезпечення.

Інтеграція інструментів розробки означає спільне використання та взаємодію різних інструментів та сервісів під час процесу розробки програмного забезпечення. Це дозволяє розробникам працювати більш ефективно та зменшити зусилля на вирішення завдань. Інтеграція з системами контролю версій, такими як Git або SVN, дозволяє розробникам відстежувати зміни в коді, вносити правки паралельно та об'єднувати їх. ICP та текстові редактори можуть інтегруватися з іншими інструментами, такими як системи контролю версій, компілятори, відладчики та інші, спрощуючи робочий процес розробки. Інтеграція з системами збирання, такими як Apache Maven, Gradle чи npm, дозволяє автоматизувати процес збирання та залежностей проекту.

Підключення до систем тестування, таких як JUnit чи pytest, дозволяє автоматизувати виконання тестів та відстежувати їхні результати. Інтеграція з інструментами комунікації, такими як Slack, Microsoft Teams або e-mail, дозволяє розробникам отримувати сповіщення про зміни в коді, результати збірки чи інші події. Інтеграція з інструментами аналізу коду, такими як SonarQube або ESLint, дозволяє виявляти та виправляти потенційні проблеми в коді. Інтеграція з CI/CD-системами, такими як Jenkins, Travis CI чи GitLab CI, дозволяє автоматизувати процеси збирання, тестування та розгортання.

Інтеграція інструментів розробки сприяє підвищенню продуктивності розробницького колективу та забезпечує швидку та ефективну роботу над проектами програмного забезпечення.

Допоміжні інструменти у розробці програмного забезпечення. Графічні редактори, такі як Adobe Photoshop або GIMP, використовуються для створення та редагування графіки, іконок та інших елементів інтерфейсу користувача. Дизайн-системи, такі як Sketch або Figma, допомагають створювати та управляти дизайном інтерфейсу, а також спрощують співпрацю між дизайнерами та розробниками. Менеджери завдань, такі як Jira, Trello або Asana, допомагають організовувати

робочі завдання, визначати пріоритети та відстежувати прогрес виконання проекту. Інструменти для автоматизованого тестування, такі як Selenium або Appium, дозволяють автоматизувати виконання тестів для перевірки функціональності та взаємодії програми.

Профайлери, такі як VisualVM або Xcode Instruments, використовуються для визначення та оптимізації продуктивності програмного коду. Генератори документації, такі як Javadoc чи Sphinx, автоматизують процес створення та оновлення технічної документації для проектів. Інструменти аналізу коду, такі як SonarQube чи ESLint, допомагають виявляти потенційні проблеми в програмному коді та покращують його якість. Інструменти автоматизації деплою, такі як Ansible або Docker, спрощують розгортання та конфігурування програмного забезпечення на серверах. Інструменти моніторингу, такі як Prometheus чи ELK Stack, допомагають відстежувати та аналізувати роботу програми в реальному часі. Інструменти, такі як MySQL Workbench чи pgAdmin, дозволяють взаємодіяти з базами даних, виконувати запити та відстежувати стан даних.

Ці допоміжні інструменти сприяють розширенню можливостей розробників, полегшують робочий процес та покращують загальну продуктивність у розробці програмного забезпечення.

Фреймворки для розробки — це комплексні структури, що надають набір готових компонентів, бібліотек та інструментів для швидкої і ефективної розробки програмного забезпечення.

Django — це високорівневий фреймворк для розробки веб-додатків на мові програмування Python. Він базується на концепції "батерій включено", що означає наявність вбудованих компонентів та бібліотек для вирішення різних завдань розробки. Django спрощує створення складних веб-додатків, дозволяючи розробникам фокусуватися на бізнес-логіці та високорівневому дизайні. Django

використовує архітектурний підхід MVC, де моделі представляють дані, види відповідають за відображення та взаємодію з користувачем, а контролери керують логікою додатку. Django має вбудовану ORM, яка дозволяє взаємодіяти з базою даних за допомогою об'єктів Python, спрощуючи операції з базою даних та забезпечуючи підтримку різних СУБД. Django надає готовий адміністративний інтерфейс для керування даними в базі даних. Це дозволяє легко створювати, оновлювати та видаляти дані без написання власного адміністративного інтерфейсу. Django надає готові рішення для автентифікації користувачів та управління їхніми правами доступу, що спрощує створення системи користувачів для ваших додатків. Використання URL-шаблонів дозволяє визначити структуру URL-адрес та вказати, які види повинні бути викликані для різних запитів. Django має вбудовані засоби для захисту від різних видів атак, таких як захист від CSRF-атак, SQL-ін'єкцій та інших. Використання шаблонізатора Django дозволяє розробникам легко вбудовувати дані в HTML-сторінки та створювати динамічний вміст. Використання Django зручно для проектів будь-якої складності, від невеликих веб-сайтів до великих корпоративних систем.

Ruby on Rails — це високорівневий фреймворк для розробки веб-додатків на мові програмування Ruby. Він ґрунтується на принципах конвенції перед конфігурацією (Convention over Configuration) та реалізує архітектурний підхід Model-View-Controller (MVC). Rails розроблено для спрощення та прискорення розробки, надаючи широкий спектр готових рішень. RoR має заздалегідь визначені конвенції, що спрощує розробку. Розробники повинні вказувати лише ті налаштування, які відрізняються від стандартних. Rails використовує архітектурний підхід Model-View-Controller, де моделі відповідають за роботу з даними, види — за представлення та відображення, а контролери — за логіку додатку. RoR включає вбудований ORM-інструмент під назвою Active Record. Він

дозволяє взаємодіяти з базою даних за допомогою об'єктів, що спрощує операції з базою даних.

Роутинг в Rails зорієнтований на RESTful підхід, що дозволяє зручно визначати шляхи та контролери для обробки різних видів запитів. Rails має готові інструменти для створення стандартних елементів, таких як моделі, контролери, види тощо. Генератори коду допомагають автоматизувати рутинні завдання. Rails вбудованою має безпекові механізми, такі як захист від атак Cross-Site Scripting (XSS) та SQL Injection. Для керування даними в базі даних використовується вбудований адміністративний інтерфейс, що спрощує роботу з даними в режимі реального часу. Rails легко інтегрується з різними front-end фреймворками, такими як React або Vue.js, дозволяючи розробникам працювати з сучасними технологіями. Rails — це потужний та популярний фреймворк для розробки веб-додатків, який дозволяє створювати ефективні та сучасні рішення.

Spring — це великий екосистемний фреймворк для розробки Java-додатків. Заснований на принципах інверсії управління та аспектно-орієнтованого програмування, Spring надає багато функціоналу для вирішення різних аспектів розробки, включаючи управління бізнес-логікою, з'єднання з базою даних, обробку транзакцій, аспекти безпеки та багато іншого. Spring використовує принцип інверсії управління, де об'єкти вибудовуються та налаштовуються контейнером Spring, а не розробником. Це спрощує управління залежностями та підтримує легкість тестування. Spring підтримує аспектно-орієнтоване програмування, що дозволяє визначати аспекти бізнес-логіки, такі як логування, транзакції та безпеку, окремо від основної логіки програми. Spring розділений на ряд модулів (Spring Core, Spring MVC, Spring Data, тощо), що дає можливість вибирати тільки ті частини, які необхідні для конкретного проекту. Фреймворк легко розширюється за допомогою власних модулів.

Spring Boot — це проект, який спрощує створення автономних, легко налаштовуваних додатків на основі Spring. Він має вбудовані налаштування за замовчуванням та дозволяє швидко розпочати роботу над новим проектом. Spring Data дозволяє легко взаємодіяти з різними системами управління базами даних, надаючи загальний інтерфейс для доступу до даних. Spring Security надає інструменти для забезпечення додатків, включаючи аутентифікацію, авторизацію та захист від різних видів атак. Spring MVC — це модуль для розробки веб-додатків, який надає модель-вид-контролер архітектуру, а також підтримує RESTful веб-сервіси. Spring залишається одним з найпопулярніших фреймворків для розробки Java-додатків і надає широкий спектр можливостей для побудови робустних та масштабованих додатків.

Express.js — це легкий та гнучкий веб-фреймворк для розробки веб-додатків на платформі Node.js. Він дозволяє швидко створювати серверні додатки та веб-сайти, забезпечуючи базовий набір функціоналу для обробки HTTP-запитів, визначення маршрутів та взаємодії з базою даних. Express.js має мінімальний набір функцій, що робить його дуже гнучким. Розробники можуть вибирати додаткові бібліотеки для розширення функціоналу. Express дозволяє легко визначати маршрути для обробки HTTP-запитів. Використання маршрутів дозволяє логічно групувати та обробляти запити для різних шляхів. Middleware — це функції, які обробляють HTTP-запити та відповіді в ході їх проходження через додаток. Express надає можливість використовувати готові та власні middleware для виконання різних завдань, таких як обробка сесій, логування, обробка помилок тощо.

Express спрощує роботу з JSON-даними, що робить його ідеальним вибором для створення API. Express підтримує використання різних шаблонізаторів (наприклад, EJS, Pug), що дозволяє легко створювати HTML-сторінки. Можливість використовувати систему видів для створення сторінок та розширення функціоналу за допомогою макетів. За допомогою Express можна легко

використовувати WebSocket, що дозволяє створювати інтерактивні та реально-часові додатки. Express є важливим інструментом для розробки серверних додатків на Node.js та широко використовується для створення API, веб-сайтів та інших веб-додатків.

Laravel — це високорівневий веб-фреймворк, розроблений для полегшення та прискорення процесу створення сучасних веб-додатків на мові програмування PHP. Laravel спрощує розробку, надаючи елегантну синтаксис та ряд інструментів для вирішення різних завдань, включаючи роботу з базою даних, маршрутизацію, аутентифікацію, кешування та багато іншого. Eloquent — це ORM (Об'єктно-реляційне відображення) в Laravel, яке дозволяє взаємодіяти з базою даних за допомогою об'єктів та моделей, спрощуючи операції з базою даних. Blade — легкий та елегантний шаблонізатор, що використовується в Laravel для вбудовування PHP-коду в HTML та розширення шаблонів. Laravel використовує архітектурний підхід Model-View-Controller (MVC), де моделі представляють дані, види відповідають за відображення та взаємодію з користувачем, а контролери керують логікою додатку.

Маршрути в Laravel дозволяють визначати, які контролери та методи повинні відповідати на різні HTTP-запити. Laravel надає готовий функціонал для реалізації аутентифікації користувачів та управління їхніми правами доступу. Artisan — це консольна команда для автоматизації різних завдань, таких як створення міграцій, контролерів, моделей та інше. Laravel використовує міграції для контролю версій бази даних, а також сіди для наповнення бази даних початковими даними. Middleware в Laravel дозволяє вставати в потік обробки HTTP-запитів і виконувати певні дії, такі як перевірка автентифікації, логування тощо. Laravel — це потужний та елегантний фреймворк для розробки веб-додатків на PHP, який використовують для широкого спектру проектів, від невеликих веб-сайтів до великих корпоративних систем.

Flask — це легкий та гнучкий мікрофреймворк для розробки веб-додатків на мові програмування Python. Він відомий своєю простотою, легкістю використання та розширюваністю. Flask не накладає великого обсягу стандартів і правил, що дає розробникам велику свободу у виборі інструментів та підходів. Flask є мікрофреймворком, що означає, що він надає лише основні елементи для розробки веб-додатків, залишаючи велику частину вибору та налаштувань розробникам. Flask легко інтегрується з різними бібліотеками та інструментами, такими як SQLAlchemy для роботи з базою даних, Jinja2 для шаблонізації та WTForms для обробки форм. Flask використовує Jinja2 для шаблонізації, що дозволяє легко вбудовувати дані в HTML та інші сторінки.

Маршрутизація в Flask використовує декоратори Python, що робить визначення шляхів та функцій-контролерів простим та зрозумілим. Flask дозволяє розширювати функціональність за допомогою різноманітних розширень (extensions) або власних бібліотек та модулів. Flask підтримує RESTful розробку та надає зручність визначення ресурсів та їхніх операцій. Flask має вбудований розробницький сервер, що дозволяє швидко перевірити та випробувати додаток на локальній машині. Flask — це відмінний вибір для розробки простих та середньорозмірних веб-додатків на Python.

Існує багато фреймворків для юніт-тестування, які допомагають розробникам перевіряти правильність роботи окремих частин коду.

Фреймворки для Юніт-тестів:

JUnit — це фреймворк для тестування Java-програм, який дозволяє розробникам створювати та виконувати юніт-тести. Використовується для перевірки правильності роботи окремих методів, класів або пакетів коду. JUnit надає аннотації та класи для створення тестів та перевірки очікуваних результатів.

pytest — це потужний фреймворк для тестування Python, який пропонує простий та ефективний спосіб писати тести. Одна з основних переваг — читабельний синтаксис, який дозволяє розробникам легко організовувати та виконувати тести. pytest автоматично виявляє та виконує тести, що робить його дружелюбним для користувачів.

NUnit — це фреймворк для тестування програм на платформі .NET. Він дозволяє розробникам створювати та виконувати тести для .NET-проектів. NUnit надає розширений функціонал для організації тестів, включаючи параметризовані тести, асинхронне тестування та багато іншого.

Mocha — це фреймворк для тестування JavaScript, який часто використовується для тестування Node.js-додатків. Mocha надає гнучкість у виборі асертів, включає підтримку асинхронних тестів та розширюється за допомогою плагінів.

PHPUnit — це фреймворк для тестування PHP-коду, розроблений для використання в юніт-тестах та тестування функціональності. PHPUnit надає ряд асертів та можливостей для організації тестів, включаючи мокування об'єктів та тестові декоратори.

Ці фреймворки є стандартами у своїх мовах програмування та допомагають розробникам ефективно тестувати свій код, забезпечуючи високий рівень якості та стабільність програмних продуктів.

Фреймворки для Інтеграційних тестів:

TestNG — це фреймворк для тестування Java, який підтримує різні рівні тестування, включаючи інтеграційні тести. Забезпечує аннотації для позначення тестових методів та конфігураційних параметрів.

Cucumber — це інструмент для виконання тестів, написаних у формі "живого" тексту (Gherkin syntax). Використовується для опису поведінки та взаємодії між компонентами системи.

RestAssured — це бібліотека для автоматизованого тестування RESTful API в Java. Дозволяє легко виконувати інтеграційні тести для взаємодії з API та перевірки відповідей.

Supertest — це бібліотека для тестування веб-додатків на Node.js, яка надає можливості для виконання інтеграційних тестів HTTP-запитів та перевірки результатів.

RestSharp — це бібліотека для тестування RESTful API в .NET-проектах. Дозволяє створювати та виконувати інтеграційні тести для HTTP-запитів.

Spring Boot Test — частина фреймворка Spring, яка дозволяє легко писати тести для Spring Boot додатків, включаючи інтеграційні тести.

Ці фреймворки спрощують процес написання, виконання та аналізу інтеграційних тестів, допомагаючи забезпечити високу якість взаємодії між компонентами програмного забезпечення. Вибір конкретного фреймворку залежить від потреб та технічних особливостей вашого проекту.

Фреймворки для Відкритого тестування API:

Postman — це не лише інструмент для ручного тестування API, але й платформа для автоматизації тестування API. Вона має інтуїтивний інтерфейс, можливості скриптування за допомогою JavaScript, та можливості автоматизації колекцій тестів.

Requests — це бібліотека для тестування HTTP-запитів у Python. Це зручний інструмент для автоматизації взаємодії з API та перевірки відповідей.

SoapUI — це відомий фреймворк для тестування веб-служб, який підтримує тестування як RESTful, так і SOAP-протоколів. Він дозволяє легко створювати, виконувати та аналізувати тести для веб-служб.

Karate DSL — це фреймворк для тестування API на основі DSL, який дозволяє писати тести на мові, схожій на Gherkin. Він спеціалізується на тестуванні API та може виконувати інтеграційні тести.

Chakram — це бібліотека для тестування API на JavaScript, яка базується на Mocha та Chai. Забезпечує зручний інтерфейс для виконання запитів та перевірки результатів.

Ці фреймворки допомагають забезпечити коректну та надійну взаємодію з веб-службами та API, автоматизуючи процес тестування та забезпечуючи швидку зворотню зв'язок у випадку виявлення проблем.

2.2 Аналіз даних та бізнес-інтелект

Аналіз даних – це процес перевірки, чистки, обробки та інтерпретації даних з метою виявлення корисної інформації, визначення патернів, виявлення тенденцій та прийняття обґрунтованих рішень. Цей процес може бути застосований в різних областях, включаючи бізнес, науку, медицину та інші галузі.

Збір даних - це процес отримання інформації з різних джерел з метою подальшого аналізу, використання та прийняття обґрунтованих рішень. Цей процес є ключовим у великих і маленьких компаніях, наукових дослідженнях, галузевих аналізах та інших областях.

Етапи Збору Даних:

- Визначення Мети. Кларифікація конкретних цілей та питань, які мають вирішуватися зібраними даними.
- Розробка Плану Збору Даних. Створення стратегії та плану збору даних, включаючи вибір методів та інструментів.
- Вибір Джерел Даних. Визначення джерел, з яких будуть зібрані дані, таких як бази даних, опитування, сенсори, веб-скрапінг тощо.
- Розробка Інструментів та Засобів. Створення або вибір інструментів для збору та обробки даних, які відповідають потребам проекту.
- Розробка Опитувальних Форм. Якщо використовуються опитування, розробка опитувальних форм з урахуванням поставлених завдань.
- Створення Матриць та Планів Збору. Визначення структури та частоти збору даних, створення матриць для реєстрації інформації.

Методи та Інструменти Збору Даних:

- Опитування. Збір даних шляхом ставлення питань учасникам. Опитувальні платформи (Google Forms, SurveyMonkey), стандартні анкети.
- Спостереження. Вивчення та реєстрація дій, які відбуваються в реальному часі. Записи, відеоматеріали, спостереження.
- Аналіз Соціальних Мереж. Вивчення активності та взаємодії в соціальних мережах. API соціальних мереж, інструменти аналізу соціальних мереж.
- Сенсорні Дані. Збір інформації від сенсорів, таких як GPS, акселерометри, датчики. Мобільні пристрої, IoT-девайси.
- Веб-скрапінг. Автоматизований збір даних з веб-сайтів. Python (Beautiful Soup, Selenium), спеціалізовані інструменти.
- Аналіз Біг-Даних. Обробка та аналіз великих обсягів даних. Apache Hadoop, Apache Spark.

Виклики та Рекомендації:

- Конфіденційність та Захист Даних. Забезпечення конфіденційності та безпеки зібраних даних. Використання шифрування, захищених підключень та дотримання нормативів безпеки даних.
- Вибірка та Змішування Даних. Вибірка представляє вибірку групи учасників або даних, що може впливати на результати. Забезпечення репрезентативності вибірки, врахування можливих перекосів.
- Якість та Точність Даних. Забезпечення якості та точності зібраних даних. Використання перевірених джерел, перевірка даних на відповідність стандартам.
- Етичні Аспекти. Збирання та використання даних. Забезпечення згоди учасників, дотримання принципів етики та правил обробки особистих даних.
- Правові Аспекти. Врахування вимог законодавства щодо збору та обробки даних. Поважання законів про захист даних та конфіденційності.

Ефективний збір даних є ключовим етапом для успішного аналізу та використання інформації у різних контекстах.

Очищення та перевірка даних – це важливі етапи в аналізі даних, спрямовані на виявлення та виправлення помилок, аномалій та непропущених значень, щоб забезпечити якість та надійність даних для подальшого використання.

Головні аспекти очищення даних. Визначення обсягу та розподілу пропущених значень, а також виправлення чи видалення їх. Визначення та обробка аномальних значень, які можуть впливати на результати аналізу. Приведення даних до єдиного формату (наприклад, дати, числа). Виявлення та видалення дублікатів у наборі даних. Зміна типу даних для полегшення обробки та аналізу.

Головні аспекти перевірки даних. Визначення основних статистичних показників (середнє значення, медіана, варіація) для оцінки розподілу даних. Аналіз взаємозв'язків між різними змінними у наборі даних. Використання візуалізацій (графіків, діаграм, гістограм тощо) для виявлення патернів та аномалій. Визначення та перевірка на відповідність теоретичному розподілу. Визначення та перевірка даних на відповідність бізнес-логіці та правилам. Аналіз даних для виявлення викидів та виняткових ситуацій.

Очищення та перевірка даних є фундаментальним етапом у впорядкуванні та підготовці даних для подальшого використання у аналізі та прийнятті рішень.

Візуалізація даних – це процес представлення інформації у вигляді графіків, діаграм, графіків, карт та інших візуальних елементів для легкості розуміння, виявлення патернів та отримання інсайтів. Вона грає важливу роль у сприйнятті та аналізі даних.

Мета Візуалізації Даних:

- Посилення зрозуміння. Зробити дані більш доступними та зрозумілими для широкого аудиторії.
- Виявлення патернів та тенденцій. Відзначити ключові патерни та тенденції в даних.
- Підтримка прийняття рішень. Надати відомості, які сприяють прийняттю обґрунтованих рішень.
- Ефективна комунікація. Передати інформацію таким чином, щоб її легко сприймали інші люди.

Типи графіків та діаграм:

- Лінійні графіки. Відображення змін значень відносно часу або іншої змінної.

- Колонкові та стовпчасті графіки. Порівняння кількісних значень між категоріями.
- Кругові та донат-діаграми. Показ часток від загальної суми.
- Теплові карти. Відображення значень на сітці кольорами для виявлення взаємозв'язків.
- Графіки розсіювання. Вивчення залежностей між двома змінними.

Принципи ефективної візуалізації:

- Простота та зрозумілість. Вибір простих форм та кольорів для полегшення сприйняття.
- Відповідність завданню. Вибір типу графіку, що найкраще передає необхідну інформацію.
- Використання коректних масштабів. Забезпечення відповідності масштабу графіка суті даних.
- Використання кольорів зрозуміло. Використання чітких та відмінних кольорів зробить візуалізацію більш зрозумілою.
- Інтерактивність. Додавання елементів інтерактивності для полегшення взаємодії з даними.

Експлоративний аналіз даних (EDA) – це процес вивчення та аналізу набору даних з метою отримання візуального та кількісного розуміння їхньої структури та властивостей. Основні кроки EDA включають завантаження даних, перевірку пропущених значень, визначення розподілу даних та взаємозв'язку між змінними. Мета - зрозуміти основні характеристики даних та виявити закономірності, які можуть вказати на можливі напрямки подальшого детального аналізу. EDA допомагає виявити гіпотези, перевірити якість даних та підготувати дані для подальших досліджень.

Починається з завантаження та перегляду перших рядків даних для отримання загального уявлення про їхню структуру та зміст. Аналіз наявності та обробка пропущених значень. Визначення стратегії їхнього заповнення чи видалення. Розрахунок основних статистичних характеристик для кількісних змінних, таких як середнє значення, медіана, мода. Створення графічних представлень даних, таких як гістограми, ящикові графіки, для візуального аналізу їхнього розподілу.

Дослідження кореляцій та залежностей між різними змінними для виявлення можливих взаємозв'язків. Виявлення аномалій, викидів або несподіваних відхилень, які можуть впливати на аналіз. Формулювання гіпотез та розробка стратегій для їх перевірки або спростування. EDA дозволяє досліджувати дані на попередньому етапі, виявляти їхні особливості та структуру, щоб краще зрозуміти контекст перед більш складним аналізом чи моделюванням. Важливим є комбінація кількісного та візуального підходу для повного розуміння набору даних.

Моделювання та прогнозування в сфері аналізу даних визначають процес створення та використання моделей для прогнозування майбутніх подій, трендів або значень на основі наявних даних. Цей етап аналізу є ключовим для вирішення бізнес-задач та прийняття обґрунтованих рішень. Початковий етап, де встановлюється конкретна мета аналізу та прогнозування. Це може бути передбачення продажів, класифікація об'єктів чи інше завдання, визначене бізнес-задачами. Обрання та очищення даних для використання в моделі. Включає в себе обробку пропущених значень, перетворення категоріальних даних та інші маніпуляції для забезпечення якості вхідних даних.

Визначення підходящого методу чи алгоритму для розв'язання конкретної задачі. Це може бути лінійна регресія для прогнозування числових значень чи дерево рішень для класифікації. Розбиття даних на навчальний та тестовий набори. Навчання моделі проводиться на навчальному наборі, а тестування - на окремому

для оцінки її точності. Використання навчального набору для налаштування параметрів моделі. Це включає в себе процес оптимізації, де модель навчається адаптуватися до особливостей даних. Використання тестового набору для оцінки ефективності моделі. Метрики, такі як точність, чутливість чи специфічність, використовуються для визначення якості прогнозу.

Вносити зміни у модель або її параметри для поліпшення результатів, якщо це необхідно. Цей процес може включати зменшення перенавчання або додавання нових функцій. Використання оптимізованої моделі для створення прогнозів для нових даних або подій. Це може бути передбачення майбутніх значень чи класифікація нових об'єктів. Постійна валідація моделі за допомогою нових даних та її моніторинг для переконання в актуальності та ефективності прогнозів.

Моделювання та прогнозування є ключовим етапом в аналізі даних, що дозволяє ефективно використовувати отриману інформацію для прийняття обґрунтованих рішень та передбачення майбутніх подій. Розуміння та вірне застосування алгоритмів та інструментів грають важливу роль у вдосконаленні якості прогнозів.

Процес виявлення статистичних або кількісних зв'язків між різними змінними в наборі даних. Це може включати в себе вивчення того, які змінні взаємодіють або взаємодіють між собою. Вивчення форми та характеру розподілу різних змінних. Наприклад, чи є різниця у розподілі вартостей між різними категоріями. Згрупування даних за певною характеристикою та аналіз показників в межах кожної групи. Використання кольорових діаграм для візуального виявлення залежностей між двома чи більше змінними.

Статистичний аналіз є важливим інструментом в області аналізу даних, допомагаючи виявити закономірності, отримати інсайти та підтверджувати гіпотези. Цей процес включає в себе використання різноманітних статистичних

методів для опису, вивчення та інтерпретації даних. Ключові елементи статистичного аналізу включають в себе розрахунок основних статистичних характеристик, таких як середнє значення, медіана, мода, дисперсія та стандартне відхилення. Ці показники надають загальне уявлення про структуру та властивості даних. Забезпечує засоби для роботи з великими обсягами даних та висновками на підставі вибірок. Включає в себе побудову довірчих інтервалів, проведення гіпотез та аналіз варіацій. Дозволяє вивчати взаємозв'язок між залежною та незалежними змінними. Регресійні моделі допомагають прогнозувати значення змінної на підставі інших змінних.

Аналіз дисперсії використовується для порівняння середніх значень між трьома чи більше групами та визначення впливу факторів на змінні. Тестування гіпотез дозволяє визначити, чи можна робити висновки про всю популяцію на підставі вибірки та перевіряти припущення. Статистичний аналіз надає об'єктивні дані для прийняття обґрунтованих рішень в бізнесі, науці чи суспільстві. Статистичні методи дозволяють підтверджувати чи спростовувати гіпотези та проводити експерименти з підтримкою даних. Порівняння груп, методів чи обставин за допомогою статистичних технік для виявлення різниць та схожостей. Статистичний аналіз важливий для отримання об'єктивних висновків з даних та використовується в різних галузях для прийняття обґрунтованих рішень та вивчення взаємозв'язків між різними змінними.

2.3 Штучний інтелект та машинне навчання

Штучний інтелект — це галузь інформатики, яка займається створенням програм та систем, здатних виконувати завдання, які вимагають інтелектуальної обробки інформації. Основною метою штучного інтелекту є створення

комп'ютерних систем, які можуть мислити, вчитися та приймати рішення, схожі на людські.

Машинне навчання — це підгалузь штучного інтелекту, яка вивчає алгоритми та моделі, які дозволяють комп'ютерам вчитися та вдосконалювати свою продуктивність без явного програмування. Головна ідея полягає в тому, щоб комп'ютерні системи автоматично набували знань і вмінь на основі даних та досвіду.

Обробка природної мови (NLP) є ключовою галуззю в сфері штучного інтелекту, яка відкриває широкий спектр можливостей для взаємодії між комп'ютерами та людьми через мовний інтерфейс. Основні задачі, такі як розпізнавання та розуміння мови, машинний переклад, генерація тексту та аналіз настрою тексту, дозволяють створювати інтелектуальні системи здатні взаємодіяти з текстовою інформацією.

Однією з ключових переваг NLP є можливість автоматизації обробки текстової інформації в реальному часі. Це використовується в різноманітних застосуваннях, таких як машинний переклад для подолання мовних бар'єрів, створення інтерактивних чат-ботів для обслуговування клієнтів, аналіз великих обсягів текстових даних для виявлення патернів та здійснення прогнозів.

Однак, важливо враховувати виклики, пов'язані з NLP, такі як складність аналізу варіативності мови, розпізнавання індивідуального смислу та збереження конфіденційності текстових даних. Для досягнення успіху у цій галузі, необхідно поєднувати технічні знання з розумінням контексту та особливостей мови.

NLP продовжує розвиватися, і разом з іншими технологіями штучного інтелекту створює нові можливості для поліпшення ефективності бізнес-процесів, забезпечення комфортної взаємодії користувачів та розширення можливостей аналізу текстової інформації.

Комп'ютерне зорове сприйняття є важливою та швидкозростаючою галуззю в сфері штучного інтелекту. Здатність комп'ютерів розпізнавати та розуміти візуальні дані, такі як зображення та відео, відкриває широкий спектр можливостей в різноманітних областях.

Однією з ключових переваг комп'ютерного зорового сприйняття є його застосування в реальному часі для різноманітних завдань. Від розпізнавання облич та об'єктів до медичних діагнозів та автономних автомобілів, ця технологія знаходить своє застосування в різних галузях.

Процес розвитку комп'ютерного зорового сприйняття включає в себе вдосконалення алгоритмів машинного навчання, використання нейронних мереж та розширення наборів даних для тренування. Однак важливо враховувати виклики, пов'язані з етичними питаннями, конфіденційністю даних та необхідністю підтримки високої точності розпізнавання.

Майбутнє комп'ютерного зорового сприйняття обіцяє ще більше інновацій та застосувань. З розвитком технологій та зростанням обчислювальної потужності можна очікувати подальший прогрес у вдосконаленні систем візуального розпізнавання та розширенні їхнього впливу на різні галузі життя.

Робототехніка, яка використовує технології штучного інтелекту та мехатроніки для створення інтелектуальних роботів та автоматизованих систем, є надзвичайно важливою галуззю, що перетинає багато галузей науки і техніки. Вона визначає та формує майбутнє, де технічні здобутки роблять роботи більш ефективними та ергономічними.

Використання роботів в різних сферах, таких як виробництво, медицина, наука та побутове використання, значно підвищує ефективність та точність виконання завдань. Роботи здатні виконувати рутинні операції, а також взаємодіяти з людьми у безпечних та продуктивних способах.

Однак існують виклики, пов'язані з етичними аспектами використання роботів, включаючи питання безпеки, вплив на ринок праці та прозорість в прийнятті рішень роботами. Щоб ефективно впроваджувати робототехніку, необхідно розробляти стандарти та регулювати використання роботів у різних сферах.

За відсутності сумніву, робототехніка залишається однією з ключових галузей, яка формує та перетворює сучасний світ. Продовжуючи досліджувати та вдосконалювати технології робототехніки, ми можемо сприяти створенню більш безпечного, ефективного та інтегрованого суспільства.

Однією з ключових переваг машинного навчання є можливість автоматичної адаптації до змінних умов і навчання на основі даних. Алгоритми машинного навчання дозволяють системам аналізувати великі обсяги інформації та виявляти складні патерни, що робить їх ефективними інструментами для прийняття рішень.

Проте, разом з перевагами існують і виклики, такі як необхідність великих обсягів даних для тренування, прозорість та відповідальне використання алгоритмів. Ефективне використання машинного навчання вимагає уважного вивчення та врахування етичних та соціальних аспектів.

Машинне навчання продовжує активно розвиватися, і його вплив на технологічний ландшафт обрізається в усіх сферах життя. Те, як ми управляємо та розвиваємо цю технологію в майбутньому, визначить напрямок нашого технологічного розвитку та його вплив на суспільство.

Лінійна регресія є статистичним методом, що використовується для моделювання лінійного відношення між залежною змінною (вихідним значенням) і однією чи декількома незалежними змінними (вхідними факторами). Метою лінійної регресії є побудова лінії, яка найкращим чином підходить до набору даних.

У лінійній регресії використовується лінійна функція для моделювання відношень між змінними. Для одновимірного випадку це може бути виражено рівнянням: $y = mx + b$, де y – залежна змінна, x – незалежна змінна,m – нахил лінії(коефіцієнт наклону),b – константа(відсув або вільний член).

У лінійній регресії використовується метод найменших квадратів для знаходження лінії, яка мінімізує квадратичну різницю між реальними та передбаченими значеннями. Це дозволяє знайти оптимальні значення параметрів моделі.

Штучні нейрони є базовими будівельними блоками нейронних мереж і відіграють ключову роль у функціонуванні цих мереж. Їхнє моделювання підприємець у великій мірі імітує функції нейронів у людському мозку, де інформація обробляється за допомогою зв'язків між нейронами.

Основні характеристики штучних нейронів включають вхідні сигнали, ваги, зсуви та функції активації. Колективне використання нейронів в шарах різних типів (вхідний, прихований, вихідний) дозволяє нейронним мережам вирішувати різноманітні завдання, від класифікації до генерації тексту.

Штучні нейрони знаходять широке використання в різних сферах, включаючи обробку зображень, мовний аналіз, прогнозування та оптимізацію. Трендом у сучасних дослідженнях та застосуваннях є використання глибоких нейронних мереж, які володіють великою потужністю у розв'язанні складних завдань.

Важливими викликами залишаються потреба у великих обсягах даних для ефективного тренування, прозорість та визначення етичних стандартів у використанні нейронних мереж. Однак їхній внесок у розвиток штучного інтелекту та розв'язання реальних проблем сучасного світу робить їх важливим інструментом в галузі машинного навчання.

Висновки до розділу 2

У цьому розділі ми детально розглянули сучасні програмні інструменти, які використовуються у розробці програмного забезпечення. Цей аналіз вказує на велику різноманітність та інноваційність інструментарію, який розробники використовують для створення високоякісних та ефективних програм.

Інтегровані середовища розробки (IDE) надають зручне середовище для написання коду, відлагодження та тестування програм. Ми докладно розглянули такі інструменти, як Visual Studio Code та IntelliJ IDEA, які отримали популярність завдяки своїй легкості використання та розширюваності.

Текстові редактори відіграють важливу роль у швидкому та зручному редагуванні коду. Sublime Text та Visual Studio Code відзначаються своєю легкістю та потужністю, роблячи їх популярними серед розробників.

Компілятори та інтерпретатори визначають ефективність виконання коду. Ми розглянули використання Java та Kotlin для розробки Android-додатків, а також Swift для розробки для платформи iOS.

Відладчики є невід’ємною частиною розробки, допомагаючи ідентифікувати та виправляти помилки у програмному коді. Використання відладчиків, таких як Android Studio Debugger та Xcode Debugger, робить процес відлагодження ефективним та простим.

Управління версіями є критичним елементом для забезпечення спільної роботи над проектами. Використання систем контролю версій, таких як Git та GitHub, забезпечує ефективний контроль версій та спільну роботу.

Системи підсвічування синтаксису та автодоповнення покращують продуктивність розробників, роблячи процес написання коду більш швидким та точним. Інструменти, такі як IntelliJ IDEA та Xcode, відзначаються зручним підсвічуванням синтаксису та автодоповненням.

Усі ці інструменти створюють потужне та гнучке середовище для розробників, дозволяючи їм творити інноваційне програмне забезпечення, яке відповідає вимогам сучасного ринку. Використання цих інструментів є ключовим чинником для досягнення успіху в сфері розробки програмного забезпечення.

3 ЗАСТОСУВАННЯ ТА СТВОРЕННЯ МОБІЛЬНОГО ДОДАТКУ ДЛЯ СИСТЕМ БРОНЮВАННЯ БІЛЕТІВ

3.1 Обрання IDE та мови для написання додатку

Мобільні додатки для систем бронювання білетів стали невід'ємною частиною сучасного транспортного та розважального секторів. У цьому розділі розглянемо сучасний програмний інструментарій, який використовується для розробки таких додатків, а також дослідимо основні аспекти їх застосування.

Вибір інтегрованого середовища розробки визначає ефективність та зручність процесу створення програмного забезпечення. У розділі розглядається важливість цього етапу у контексті розробки мобільних додатків для систем бронювання білетів. Android Studio обрано як основне середовище для розробки для платформи Android. Визнане як стандарт у галузі Android-розробки, Android Studio надає багатофункціональність, включаючи відлагодження, аналіз коду та інтеграцію з Android SDK.

Існує багато різних IDE для різних мов програмування та платформ. Вибір конкретної IDE залежить від потреб розробки та особистих вподобань розробника. В нашому випадку Android Studio стає перевагою з кількох ключових причин:

- **Офіційне середовище розробки для Android.** Android Studio розроблено спеціально для створення Android-додатків і є офіційним інструментом від Google. Це забезпечує високий рівень сумісності з Android-платформою та враховує всі оновлення та нововведення.
- **Широкий функціонал та інтеграція.** Android Studio надає широкий функціонал для ефективної розробки, включаючи редактор коду, відлагодження, інструменти для розробки інтерфейсу користувача,

тестування та профілювання. Інтеграція з системами контролю версій, такими як Git, дозволяє легко відстежувати зміни та спільно працювати в команді.

- **Підтримка Котлін та Java.** Android Studio повністю підтримує як Java, так і Kotlin, що відкриває можливості вибору мови програмування в залежності від ваших уподобань та потреб проекту.
- **Інтегрований емулятор та відлагоджувальник.** Вбудований емулятор Android Studio дозволяє вам перевірити та тестувати свій додаток на різних пристроях та версіях Android. Відлагоджувальник надає потужні засоби для виявлення та усунення помилок.
- **Інновації та оновлення.** Android Studio постійно оновлюється з новими функціями та інструментами, що відповідає за останніми трендами у світі мобільної розробки. Це забезпечує вас сучасними можливостями та підтримкою новітніх фреймворків.
- **Активна спільнота та документація.** Значна кількість розробників користується Android Studio, що означає широку спільноту та велику кількість документації та онлайн-ресурсів для вирішення проблем та отримання порад.

Далі обираємо мову програмування, вибір Kotlin для розробки.

Android-додатків може бути обгрунтованим з кількох причин:

- **Офіційна підтримка.** Kotlin офіційно підтримується Google для розробки Android-додатків. Це означає, що Kotlin інтегрований безпосередньо в Android Studio, надається документація та рекомендації, і має високий рівень сумісності з Android-екосистемою.
- **Сучасність та зручність.** Kotlin є сучасною мовою програмування, яка пропонує багато функцій, що роблять код більш читабельним та

компактним. Наприклад, нульові замість вказування на null, розширення функціональності та інші властивості полегшують розробку та підтримку коду.

- **Безпека та надійність.** Kotlin створений з урахуванням питань безпеки та надійності. Він пропонує вбудовані інструменти для уникнення типових помилок, що може покращити безпеку та стабільність вашого коду.
- **Інтероперабельність з Java.** Kotlin є 100% інтероперабельним з Java, що означає, що ви можете поступово переходити до Kotlin, додаючи новий код або модернізуючи існуючий, не торкаючись при цьому Java-коду. Це дає можливість поступово впроваджувати Kotlin у великі проекти або спільно працювати в команді, де використовується Java.
- **Активна спільнота та розвиток.** Kotlin має активну спільноту розробників, яка активно співпрацює над розвитком мови та створенням корисних ресурсів. Це забезпечує підтримку та обмін знаннями.
- **Рекомендації від Google.** Google активно рекомендує використовувати Kotlin для нових проектів Android, і це виражається в їхніх офіційних матеріалах та підтримці.

Для додатку для бронювання білетів важливо вибрати підходящу базу даних. Зазвичай використовують реляційні бази даних, такі як MySQL або PostgreSQL, оскільки вони добре підходять для відносно структурованих даних. Створення таблиць для сутностей, таких як користувачі, події, рейси та квитки. В кожній таблиці можуть бути колонки, які зберігають відомості про об'єкти цих сутностей. Забезпечення ефективності та цілісності даних шляхом нормалізації. Це включає розподіл даних між таблицями, щоб уникнути дублювання та забезпечити зручний доступ до інформації. Визначення запитів та процедур бази даних для виконання

операцій, таких як додавання нових рейсів, бронювання квитків, перегляд історії покупок тощо. Застосування механізмів автентифікації та авторизації для захисту даних. Кожен користувач повинен мати унікальний ідентифікатор, а доступ до певних ресурсів має бути обмеженим. Зберігання інформації про рейси, таку як місце відправлення, місце призначення, дати та часи вильотів, наявність вільних місць. Реалізація механізмів бронювання та оплати квитків. Зберігання інформації про статус бронювання та історію оплат. Забезпечення синхронізації даних між сервером та клієнтами, щоб користувачі завжди отримували актуальну інформацію. Оптимізація запитів для забезпечення ефективного використання ресурсів бази даних. Введення системи логування для відстеження дій користувачів та важливих подій в системі, щоб мати можливість аудитування. Забезпечення системи регулярного резервного копіювання для захисту від втрати даних у випадку аварії. Загальний підхід до використання бази даних в додатку для бронювання білетів полягає в створенні добре структурованої та нормалізованої бази даних, яка забезпечить ефективну роботу додатка та надійність обробки інформації.

В програмуванні "state" (стан) вказує на поточний стан об'єкта чи системи. У багатьох програмних парадигмах, таких як React (JavaScript), Flutter (Dart), SwiftUI (Swift), Vue.js (JavaScript), стан використовується для відстеження та зберігання даних, які можуть змінюватися протягом життєвого циклу програми чи компонента.

У React, наприклад, "state" є об'єктом, який визначає поведінку та відображення компонента. State може містити дані, які змінюються під час виконання програми, і, якщо стан обновлюється, React автоматично перерендерює компонент, відображаючи новий стан.

Чому використовувати state для зберігання значень:

- Використання state дозволяє автоматично перерендерювати компонент при зміні стану, що полегшує реакцію на взаємодію користувача чи зміни даних.
- State надає простий інтерфейс для керування даними в компонентах. Це сприяє підтримці структурованих та організованих даних.
- React використовує алгоритми для оптимізації процесу перерендерингу, і використання state дозволяє React ефективно визначати, коли потрібно оновлювати інтерфейс.
- Використання state спрощує управління станом великих та складних додатків.
- State надає однозначний шлях для визначення, де і як зберігаються дані в компоненті.

Загалом, використання state є потужним інструментом для роботи зі змінюваними даними в багатьох сучасних фреймворках та бібліотеках.

3.2 Співпрограми

Співпрограма або співпроцедура (англ. Coroutine) — компонент програми, що узагальнює поняття підпрограми, додатковою підтримкою безлічі точок входу (а не однієї, як підпрограма) і зупинку та продовження виконання із збереженням певного положення. Співпрограми є гнучкішими і узагальненішими, ніж підпрограми, але рідше використовуються на практиці. Застосування співпрограм, будучи методикою ще асемблера, практикувалося лише деякими високорівневими мовами, наприклад Simula і Modula-2. Співпрограми добре придатні для реалізації багатьох схожих компонентів програм (ітераторів, нескінченних списків, каналів, волокон).

Співпрограма має початкову точку входу і розташовані всередині послідовності повернень і наступних точок входу. Підпрограмма може повертати керування лише одного разу, співпрограма може повертати керування багаторазово. Час роботи підпрограми відзначається принципом LIFO (остання викликана підпрограма завершується першою), час роботи співпрограми визначається її використанням і необхідністю.

Основні поняття та підходи в контексті співпрограмного програмування. Паралельність – це характеристика системи, що дозволяє виконувати кілька задач одночасно. Це може відбуватися на багатьох рівнях, включаючи апаратний рівень (багатоядерні процесори) та програмний рівень (потоки виконання, процеси). Поток виконання – це невелика одиниця виконання, яка може виконувати інструкції програми. Зазвичай багато потоків виконання працюють паралельно у межах одного процесу. Процес – це виконуюча програма, яка має власну адресу пам'яті та власний потік виконання. Процеси можуть взаємодіяти один з одним через міжпроцесовий обмін даними.

Асинхронне програмування – це підхід, коли програма виконується без чекання завершення конкретної операції. Зазвичай використовується у контексті обробки подій та введення/виведення. Корутини (generators, async/await) можуть бути використані для структурування асинхронного коду, де обчислення можуть бути призупинені та відновлені. Розділена робота (task parallelism) включає в себе розділення завдань на невеликі підзадачі, які виконуються паралельно. Синхронізація використовується для забезпечення взаємодії між співпрограмами, щоб уникнути конфліктів та забезпечити вірність даних.

Співпрограми можна розглядати як легкі потоки, але існує ряд важливих відмінностей, які сильно відрізняють їхнє використання в реальному житті від потоків. Наступний код реалізує робочий корутин:

```
fun main() = runBlocking {  // this: CoroutineScope
    launch {  // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default
time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous
one is delayed
}
```

Давайте розберемо, що робить цей код.

launch є конструктором співпрограм. Він запускає нову співпрограму одночасно з рештою коду, який продовжує працювати незалежно. Тому **“Hello”** було надруковано першим.

delay — це спеціальна функція призупинення. Він призупиняє співпрограму на певний час. Призупинення співпрограми не блокує базовий потік, але дозволяє іншим співпрограмам запускатися та використовувати базовий потік для свого коду.

runBlocking також є конструктором співпрограм, який об’єднує світ несопрограм звичайної **fun main()** і код із співпрограмами всередині фігурних дужок **runBlocking { ... }**. Це підсвічується в середовищі IDE таким чином: **this:CoroutineScope** одразу після відкриваючої фігурної дужки **runBlocking**.

Якщо ви видалите або забудете **runBlocking** у цьому коді, ви отримаєте помилку під час виклику запуску, оскільки запуск оголошено лише в CoroutineScope.

Назва **runBlocking** означає, що потік, який його запускає (в даному випадку — основний потік), блокується на час виклику, поки всі співпрограми всередині **runBlocking { ... }** не завершать своє виконання. Ви часто побачите, що **runBlocking**

використовується таким чином на самому верхньому рівні програми та досить рідко в реальному коді, оскільки потоки є дорогим ресурсом, і їх блокування є неефективним і часто небажаним.

Співпрограми дотримуються принципу **структурованої паралельності**, що означає, що нові співпрограми можна запускати лише в певному **CoroutineScope**, який обмежує час життя співпрограми. Наведений вище приклад показує, що **runBlocking** встановлює відповідну область, і тому попередній приклад чекає **“World!”** друкується після секундної затримки і лише потім виходить.

У реальній програмі ви запускатимете багато співпрограм. Структурований паралелізм гарантує, що вони не будуть втрачені та не витоку. Зовнішню область не можна завершити, доки не завершаться всі її дочірні співпрограми. Структурований паралелізм також гарантує, що будь-які помилки в коді належним чином повідомлятимуться та ніколи не будуть втрачені.

Співпрограми менш ресурсомісткі, ніж потоки JVM. Код, який вичерпує доступну пам’ять JVM під час використання потоків, може бути виражений за допомогою співпрограм, не досягаючи обмежень ресурсів.

Наприклад, наступний код запускає 50 000 окремих співпрограм, кожна з яких чекає 5 секунд, а потім друкує крапку ('.'), споживаючи дуже мало пам'яті:

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    repeat(50_000) { // launch a lot of coroutines
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

Якщо ви напишете ту саму програму за допомогою потоків (вилучіть **runBlocking**, замініть **launch** на **thread** і замініть **delay** на **Thread.sleep**), вона споживатиме багато пам’яті. Залежно від вашої операційної системи, версії JDK і її налаштувань, він або видасть помилку браку пам’яті, або повільно запускатиме потоки, щоб ніколи не було забагато одночасно запущених потоків.

На додаток до області видимості співпрограми, наданої різними конструкторами, можна оголосити власну область за допомогою конструктора **coroutineScope**. Він створює область співпрограми і не завершується, доки не завершаться всі запущені дочірні програми.

Конструктори **runBlocking** і **coroutineScope** можуть виглядати подібно, оскільки вони обидва чекають завершення свого тіла та всіх його дочірніх елементів. Основна відмінність полягає в тому, що метод **runBlocking** блокує поточний потік для очікування, тоді як **coroutineScope** просто призупиняє, звільняючи базовий потік для інших видів використання. Через цю різницю **runBlocking** є звичайною функцією, а **coroutineScope** є функцією призупинення.

Ви можете використовувати **coroutineScope** з будь-якої функції призупинення. Наприклад, ви можете перемістити одночасний друк **“Hello”** та **“World”** у призупинену функцію **doWorld():**

```
fun main() = runBlocking {
    doWorld()
}
suspend fun doWorld() = coroutineScope {  // this: CoroutineScope
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

Конструктор **coroutineScope** можна використовувати в будь-якій функції призупинення для виконання кількох одночасних операцій. Давайте запустимо дві одночасні співпрограми всередині функції призупинення **doWorld**:

```
// Sequentially executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}
// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

Обидві частини коду всередині блоків **launch { ... }** виконуються одночасно, при цьому **“World 1”** друкується першим, через секунду від початку, а **“World 2”** друкується наступним, через дві секунди від початку. **CoroutineScope** у **doWorld** завершується лише після того, як обидва завершені, тому **doWorld** повертається та дозволяє друкувати рядок **Done** лише після цього.

3.3 Авторизація, бронювання, оплата.

Для реалізації авторизації в додатку бронювання білетів на мові програмування Kotlin можна використовувати, наприклад, систему авторизації на основі електронної пошти та пароля. Нижче наведений приклад коду для авторизації користувача за допомогою бази даних та бібліотеки для роботи з паролями BCrypt.

Додаємо бібліотеку BCrypt у проект. Додаємо наступний рядок у файл build.gradle:

```
implementation("org.mindrot:jbcrypt:0.4")
```

Створюємо клас User для представлення користувача:

```
data class User(val email: String, val passwordHash: String)
```

Створюємо клас UserRepository, який буде відповідати за роботу з базою даних для користувачів:

```
class UserRepository {
    private val users = mutableListOf<User>()

    fun createUser(email: String, password: String): User {
        val passwordHash = BCrypt.hashpw(password, BCrypt.gensalt())
        val newUser = User(email, passwordHash)
        users.add(newUser)
        return newUser
    }

    fun getUserByEmail(email: String): User? {
        return users.find { it.email == email }
    }

    fun validateUser(email: String, password: String): Boolean {
        val user = getUserByEmail(email)
        return user != null && BCrypt.checkpw(password,
user.passwordHash)
    }
}
```

Використовуємо клас UserRepository для авторизації користувача в основному класі вашого додатку:

```
fun main() {
    val userRepository = UserRepository()
    // Реєстрація нового користувача
    val newUser = userRepository.createUser("user@example.com",
"securePassword")
    // Авторизація користувача
    val email = "user@example.com"
    val password = "securePassword"

    if (userRepository.validateUser(email, password)) {
        println("Користувач авторизований успішно.")
    } else {
        println("Помилка авторизації. Невірний email або пароль.")
    }
}
```

Код можна покращити, можна використовувати більш безпечні практики, такі як використання HTTPS для передачі даних, перевірку валідності даних тощо.

Враховуючи вже створений клас **User** та **UserRepository**, давайте створимо клас **Ticket** та **BookingService.**

```
data class Ticket(val id: String, val flightName: String, val
seatNumber: String, val isBooked: Boolean = false)

class BookingService(private val userRepository: UserRepository) {
    private val tickets = mutableListOf<Ticket>()
```

Фрагмент коду нижче предствляє метод **bookTicket** класу **BookingService** і відповідає за бронювання білетів.

```
fun bookTicket(userEmail: String, flightName: String, seatNumber:
String): Ticket? {
    val user = userRepository.getUserByEmail(userEmail)
    if (user != null) {
        val ticketId = generateTicketId()
        val newTicket = Ticket(ticketId, flightName, seatNumber)
        if (!isSeatBooked(newTicket)) {
            tickets.add(newTicket)
            println("Білет успішно заброньовано для користувача
${user.email}.")
            return newTicket
        } else {
            println("Місце ${seatNumber} вже заброньовано.")
        }
    } else {
        println("Користувача з email $userEmail не знайдено.")
    }
    return null
}
```

Спочатку метод викликає **getUserByEmail** для отримання об'єкта користувача за вказаним електронним адресом (**userEmail**). Після цього виконується перевірка, чи користувач існує. Якщо **user** не дорівнює **null**, це означає, що користувач знайдений, і код виконується далі. Генерується унікальний ідентифікатор для нового білету за допомогою **generateTicketId**. Створюється новий об'єкт **Ticket** з отриманим ідентифікатором, назвою рейсу та номером місця. Викликається **isSeatBooked**, щоб перевірити, чи вже заброньовано вказане місце на рейсі. Якщо місце не заброньоване, новий білет додається до списку **tickets**, і метод

повертає створений білет. Якщо місце вже заброньоване, виводиться повідомлення про це. У випадку, якщо користувача не знайдено, виводиться відповідне повідомлення. Метод повертає **null**, якщо бронювання білету не вдалося, або повертає об'єкт **Ticket**, якщо бронювання пройшло успішно.

```
private fun isSeatBooked(ticket: Ticket): Boolean {
        return tickets.any { it.flightName == ticket.flightName &&
it.seatNumber == ticket.seatNumber && it.isBooked }
    }

    private fun generateTicketId(): String {
        return java.util.UUID.randomUUID().toString()
    }
}
```

isSeatBooked – Перевірка статусу місця. Ця функція призначена для перевірки, чи вже заброньоване місце на рейсі. Вона використовує функцію any, яка перевіряє, чи хоча б один елемент у списку tickets має такі ж властивості, як і переданий об'єкт ticket. У разі успішної броні повертає true, в іншому випадку - false.

generateTicketId – Генерація унікального ідентифікатора. Ця функція використовує **java.util.UUID.randomUUID().toString()** для створення унікального ідентифікатора білету. Проте, коментар зазначає, що у реальному додатку рекомендується використовувати більш безпечний механізм для генерації унікальних ідентифікаторів. Це може включати в себе використання бібліотек, таких як **SecureRandom** або інших безпечних методів генерації.

Код нижче демонструє основний сценарій реєстрації, авторизації та бронювання білету у консольному додатку.

```
val userRepository = UserRepository()
val bookingService = BookingService(userRepository)
```

Спочатку створюються об'єкти **UserRepository** та **BookingService**, які використовуються для роботи з користувачами та бронюванням білетів відповідно.

```
val newUser = userRepository.createUser("user@example.com",
"securePassword")
```

Викликається метод **createUser** об'єкта **userRepository** для реєстрації нового користувача з електронною поштою "**user@example.com**" та паролем **"securePassword".**

```
val email = "user@example.com"
val password = "securePassword"

    if (userRepository.validateUser(email, password)) {
        // Бронювання білету для авторизованого користувача
        val ticket = bookingService.bookTicket(email, "Flight123",
"A1")
        if (ticket != null) {
            println("Інформація про білет: ${ticket.id}, Місце:
${ticket.seatNumber}, Рейс: ${ticket.flightName}")
        } else {
            println("Бронювання білету не вдалося.")
        }
    } else {
        println("Помилка авторизації. Невірний email або пароль.")
    }
}
```

Викликається метод **validateUser** об'єкта **userRepository** для перевірки валідності введених **email** та **password**. Якщо користувач валідний, виконуються додаткові кроки, в іншому випадку виводиться відповідне повідомлення.

Викликається метод **bookTicket** об'єкта **bookingService** для спроби забронювати білет на рейс "**Flight123**" та місце "**A1**". Якщо бронювання успішне, виводиться інформація про білет, в іншому випадку виводиться повідомлення про невдале бронювання.

Якщо користувач не пройшов авторизацію, виводиться відповідне повідомлення.

Оплата в додатку для бронювання білетів може бути реалізована різними способами, такими як інтеграція з платіжними шлюзами або використання зовнішніх сервісів. Нижче подано приклад коду, який демонструє сценарій оплати.

```
data class Payment(val amount: Double, val currency: String)
class PaymentService {
    fun processPayment(payment: Payment, paymentMethod: String):
Boolean {
        // Логіка обробки платежу, можливо, використовуючи зовнішні
платіжні шлюзи або сервіси
        println("Обробка платежу ${payment.amount}
${payment.currency} за допомогою методу оплати $paymentMethod")
        // Припускаємо, що оплата завжди успішна (у реальному
застосуванні потрібно реалізувати обробку відповіді від платіжної
системи)
        return true
    }
}
class BookingService(private val paymentService: PaymentService) {
    fun bookAndPay(userEmail: String, flightName: String,
seatNumber: String, paymentMethod: String): Boolean {
        // Логіка бронювання білету
        // Припускаємо, що бронювання успішне і отримуємо інформацію
про оплату
        val paymentInfo = Payment(100.0, "USD")
        // Оплата бронювання
        val isPaymentSuccessful =
paymentService.processPayment(paymentInfo, paymentMethod)
        // Повертаємо результат операції
        return isPaymentSuccessful
    }
}
fun main() {
```

```
    val paymentService = PaymentService()
    val bookingService = BookingService(paymentService)

    val userEmail = "user@example.com"
    val flightName = "Flight123"
    val seatNumber = "A1"
    val paymentMethod = "CreditCard"
    // Бронювання та оплата
    val isBookingAndPaymentSuccessful =
bookingService.bookAndPay(userEmail, flightName, seatNumber,
paymentMethod)
    if (isBookingAndPaymentSuccessful) {
        println("Бронювання та оплата успішні.")
    } else {
        println("Бронювання або оплата не вдалася.")
    }
}
```

Приєднання бази даних до додатку включає в себе кілька ключових етапів. Давайте розглянемо загальний підхід та приклад для додатку бронювання білетів на Kotlin. У цьому прикладі буде використовована база даних SQLite, яка легко інтегрується в багато мов програмування, включаючи Kotlin.

Етапи приєднання бази даних:

- Визначте модель даних. Розробіть структуру даних, яку ви хочете зберігати у базі даних. У вашому випадку це може бути інформація про користувачі, білети та інші відповідні дані.

- Ініціалізація та налаштування бази даних. Встановіть та налаштуйте базу даних. У даному випадку буде використовована бібліотека SQLite.

- Створення класів для взаємодії з базою даних. Створіть класи для роботи з базою даних, такі як DatabaseHelper, UserDao, TicketDao і т.д. Ці класи повинні реалізовувати необхідні операції (створення таблиць, вставка, вибірка тощо).
- Інтеграція з сервісами. Використовуйте ці класи в сервісах вашого додатку (наприклад, в BookingService). Здійснюйте взаємодію з базою даних при реєстрації користувачів, бронюванні білетів і т.д.

Давайте створимо простий приклад для додавання та отримання користувача з бази даних SQLite в додатку:

```
import java.sql.Connection
import java.sql.DriverManager
import java.sql.Statement
data class User(val id: Int, val email: String, val password:
String)
class DatabaseHelper {
    private val url = "jdbc:sqlite:test.db"
    init {// Ініціалізація бази даних при створенні об'єкта
DatabaseHelper
        createTable()}
    private fun createTable() {// Створення таблиці Users при її
відсутності
        val sql = """
            CREATE TABLE IF NOT EXISTS Users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                email TEXT NOT NULL,
                password TEXT NOT NULL
            );
        """.trimIndent()
        DriverManager.getConnection(url).use { connection ->
            connection.createStatement().use { statement ->
                statement.executeUpdate(sql)
```

```
            }
        }
    }
    fun addUser(email: String, password: String): Int {// Додавання нового користувача до бази даних
        val sql = "INSERT INTO Users (email, password) VALUES ('$email', '$password');"
        DriverManager.getConnection(url).use { connection ->
            connection.createStatement().use { statement ->
                statement.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS)?.use { generatedKeys ->
                    if (generatedKeys.next()) {
                        return generatedKeys.getInt(1)
                    }
                }
            }
        }
        return -1
    }
    fun getUserByEmail(email: String): User? {
        val sql = "SELECT * FROM Users WHERE email = '$email';"
        DriverManager.getConnection(url).use { connection ->
            connection.createStatement().use { statement ->
                statement.executeQuery(sql)?.use { resultSet ->
                    if (resultSet.next()) {
                        return User(
                            id = resultSet.getInt("id"),
                            email = resultSet.getString("email"),
                            password = resultSet.getString("password")
                        )
                    }
                }
```

```
            }
        }
        return null
    }
}// Отримання користувача за email з бази даних

fun main() {
    val dbHelper = DatabaseHelper()

    // Додавання нового користувача
    val userId = dbHelper.addUser("user@example.com",
"securePassword")
    println("Доданий користувач з ID: $userId")

    // Отримання користувача за email
    val userEmail = "user@example.com"
    val user = dbHelper.getUserByEmail(userEmail)
    if (user != null) {
        println("Знайдений користувач: ${user.id}, ${user.email},
${user.password}")
    } else {
        println("Користувача з email $userEmail не знайдено.")
    }
}
```

Висновки до розділу 3

У ході створення та розгляду мобільного додатку для систем бронювання білетів, ми вивчили різні аспекти розробки, включаючи використання мови програмування Kotlin, архітектурні принципи, роботу з мобільними платформами, взаємодію з сервером через API, використання бази даних для зберігання даних

користувачів та білетів, а також розглянули питання безпеки та кращих практик в розробці мобільних додатків.

У розділі реєстрації та авторизації було використано Firebase Authentication для надання готових рішень з аутентифікації. Це дозволяє безпечно управляти обліковими записами користувачів та надає зручний інтерфейс для реєстрації та входу.

В розділі бронювання білетів була створена логіка для генерації унікальних ідентифікаторів білетів та перевірки доступності місць. Це дозволяє користувачам успішно здійснювати бронювання та уникати конфліктів. Використано клас PaymentService для обробки оплати, де можливість інтеграції реальних платіжних шлюзів була б детальною логікою обробки платежів. Тут реалізовано простий механізм для демонстрації концепції. Firebase використовувався для аутентифікації та зберігання користувачів. Це спрощує роботу з обліковими записами користувачів та надає потужні інструменти для керування даними. Взаємодія з сервером через REST API демонструє асинхронні запити для обміну даними. Забезпечується синхронізація інформації про бронювання між клієнтом та сервером.

Код був написаний, враховуючи питання безпеки, такі як використання параметризованих запитів для запобігання SQL-ін'єкціям, а також використання HTTPS для захисту передачі даних.

Створений додаток має потенціал для реального використання у системах бронювання білетів, надаючи зручний та безпечний спосіб бронювання та оплати. Проте, для реальної експлуатації необхідно враховувати широкий спектр аспектів, включаючи безпеку, ефективність та досвід користувача.

ВИСНОВКИ

Дослідження сучасного програмного інструментарію для створення мобільних додатків для систем бронювання білетів відкрило широкий спектр можливостей та технологій, які сприяють розробці високоефективних та інноваційних рішень.

Сучасні мобільні додатки для бронювання білетів використовують високопродуктивні мови програмування, такі як Kotlin та Swift, для розробки на платформах Android та iOS відповідно. Інтеграція з Firebase та іншими хмарними сервісами дозволяє забезпечити швидку та надійну роботу додатків, зокрема у сферах аутентифікації, баз даних та розсилки повідомлень.

Системи керування станом, такі як Redux або SwiftUI, дозволяють зручно та ефективно працювати зі станом додатку, забезпечуючи високу реактивність та зручний код. Корутини та асинхронні запити дозволяють оптимізувати використання ресурсів та забезпечити швидке завантаження даних.

У світлі зростання кількості користувачів мобільних додатків для бронювання білетів, особливу увагу слід звертати на забезпечення безпеки та конфіденційності даних. Використання HTTPS, шифрування даних та правильне управління автентифікацією є важливими аспектами в розробці додатків.

У майбутньому можливо спостерігати розвиток додаткових технологій, таких як розширена реальність, використання блокчейну для оплати, а також інтеграцію з розумними пристроями та системами штучного інтелекту для поліпшення досвіду користувача. Важливим є також розгляд екологічної стійкості додатків та їх впливу на навколишнє середовище.

В цілому, дослідження свідчить про активний розвиток та вдосконалення програмного інструментарію для мобільних додатків для систем бронювання

білетів, що створює обширні можливості для розробників у вдосконаленні функціональності та користувацького досвіду.

У ході створення та розгляду мобільного додатку для систем бронювання білетів, ми вивчили різні аспекти розробки, включаючи використання мови програмування Kotlin, архітектурні принципи, роботу з мобільними платформами, взаємодію з сервером через API, використання бази даних для зберігання даних користувачів та білетів, а також розглянули питання безпеки та кращих практик в розробці мобільних додатків. Ми впровадили механізми реєстрації та авторизації користувачів, забезпечуючи безпеку та конфіденційність їхніх облікових даних. Розробили сервіс бронювання білетів, використовуючи механізми генерації унікальних ідентифікаторів та перевірки доступності місць. Додали можливість оплати бронювання білетів, інтегруючи платіжні шлюзи та забезпечуючи безпечність транзакцій. Використовували Firebase для забезпечення масштабованості та простоти у використанні для аутентифікації користувачів та зберігання даних. Здійснили взаємодію з сервером для обміну даними через REST API, забезпечивши синхронізацію даних між клієнтом та сервером. Розглянули питання безпеки, включаючи захист від SQL-ін'єкцій, використання HTTPS для передачі даних, аутентифікацію та авторизацію.

Створений додаток не лише надає користувачам зручність бронювання білетів, але також демонструє ключові аспекти розробки мобільних додатків. Важливо продовжувати вдосконалювати та вдосконалювати додаток, дотримуючись сучасних стандартів розробки та забезпечуючи позитивний досвід користувача.

Щодо покращення коду можна додати більше механізмів валідації введених даних та обробки помилок, особливо при взаємодії з базою даних або сервером. При роботі з базою даних слід оптимізувати запити та використовувати індексацію для поліпшення продуктивності. Для реального застосування слід інтегрувати

реальний механізм оплати через платіжний шлюз та обробляти відповіді від нього. Підвищити зручність користування додатком шляхом додавання інтерактивних елементів та покращення дизайну. Враховуючи ці покращення, додаток буде готовий для більш широкого використання та забезпечить високий рівень функціональності та безпеки для користувачів.

Сучасні сайти та додатки для бронювання білетів можуть бути дуже функціональними та зручними для користувачів, але існує кілька аспектів, які можна подолати чи поліпшити для полегшення користування та забезпечення більшого комфорту. Більшість сайтів та додатків пропонують базові функції персоналізації, але є можливість розвивати цей аспект. Персоналізація може включати індивідуальні рекомендації, представлення улюблених напрямків або виділення акцій для конкретного користувача. Важливо, щоб додатки були оптимізовані для різних платформ (iOS, Android) та пристосовані до різних розмірів екрану, щоб забезпечити єдинообразний та зручний досвід користувача на всіх пристроях. Для спрощення процесу контролю та безпеки можна розглядати інтеграцію з технологіями розпізнавання обличчя або QR-кодами для швидшого доступу до даних та послуг. Співпраця з транспортними компаніями та використання реальних даних про перевезення може поліпшити точність інформації та забезпечити користувачам найбільш актуальну інформацію.

Додати можливість використання VR або AR для попереднього перегляду сидінь у літаку чи поїзді перед бронюванням, щоб користувачі

могли обирати оптимальні місця. Розробити інтеграцію з персональними асистентами

(наприклад, Google Assistant чи Siri), щоб користувачі могли здійснювати бронювання за допомогою голосових команд.

Застосувати технологію блокчейн для миттєвої та безпечної оплати, уникнувши проміжних етапів обробки платіжної інформації. Додати можливість

з'єднуватися з друзями та планувати подорожі разом, ділитися рекомендаціями та розкладами.

Ці нововведення можуть зробити додаток більш конкурентоспроможним та привабливим для користувачів, надаючи їм нові функції та покращений досвід.

Еволюція мобільних додатків в майбутньому буде визначатися новими технологіями, змінами в споживчих поведінках та суспільних тенденціях. Ось кілька напрямків, по яких можуть розвиватися мобільні додатки. Застосування AI та ML у мобільних додатках для індивідуалізації послуг, прогнозування потреб користувачів, оптимізації досвіду взаємодії та автоматизації завдань.

Забезпечення більшої швидкості та стабільності під час взаємодії з додатками завдяки розгортанню мереж 5G. Інтеграція з пристроями IoT для керування різними аспектами повсякденного життя, від дому та офісу до транспорту та виробництва. Використання технології блокчейн для забезпечення безпеки та прозорості в області фінансів, медицини, інтернет-торгівлі та інших галузях. Зростання уваги до кібербезпеки та заходів забезпечення приватності даних користувачів.

ПЕРЕЛІК ПОСИЛАНЬ

1. «The Pragmatic Programmer: Your Journey to Mastery» by Dave Thomas and Andy Hunt
2. «Android Programming: The Big Nerd Ranch Guide» by Bill Phillips and Chris Stewart
3. «Clean Code: A Handbook of Agile Software Craftsmanship» by Robert C. Martin
4. «The Mobile Wave: How Mobile Intelligence Will Change Everything» by Michael Saylor
5. «App: The Human Story» by Jake Schumacher
6. «Modern Web Development with Kotlin» by Hadi Hariri
7. І. Я. СПІВАК. «МЕТОДИ ТА ЗАСОБИ НАУКОВИХ ДОСЛІДЖЕНЬ В ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ». - 2012.
8. Кисіль Т.М., Зінченко О.В., Чичкарьов Є.А., Фесенко М.А.. «Алгоритмізація та програмування: Частина 1». - 2023.
9. Чичкарьов Є.А., Зінченко О.В., Фесенко М.А.. «Програмування мобільних пристроїв на Java». - 2023.
10. https://kotlinlang.org/docs/coroutines-basics.html#scope-builder
11. https://firebase.google.com/docs
12. Горбань А. Г.. «Програмування в Java». - 2008.
13. В.О. Хорошко, В.С. Чередниченко, М.Є. Шелест. «Основи інформаційної безпеки». - 2008.
14. Шикула О.М., Вишнівський В.В., Іщеряков С.М., Каргаполов Ю.В., Прокопов С.В., Щербина І.С.. «Вступ до комп’ютерного дизайну». - 2021.
15. «Effective Kotlin: 52 Specific Ways to Improve Your Use of Kotlin» by Joshua Bloch and Dianne Hackborn

16. «Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking» by Foster Provost and Tom Fawcett

ДОСЛІДЖЕННЯ СУЧАСНОГО ПРОГРАМНОГО ІНСТРУМЕНТАРІЮ СТВОРЕННЯ МОБІЛЬНИХ ДОДАТКІВ ДЛЯ СИСТЕМ БРОНЮВАННЯ БІЛЕТІВ

Студент групи КНДМ-61м

Щедрін Денис Олександрович

ЕВОЛЮЦІЯ ТЕХНОЛОГІЙ РОЗРОБКИ МОБІЛЬНИХ ДОДАТКІВ

1973 року компанія **Motorola** розробила перший мобільний апарат. Через 20 років на світ з'явився по суті перший кишеньковий комп'ютер — **Psion 3**

Ключовий період розвитку мобільних додатків, що передбачає сучасний, практично повністю пов'язаний з існуванням системи **Symbian**. **Symbian** походить від платформи **EPOC**, в результаті спільної роботи **Psion**, **Ericsson**, **Motorola** та **Nokia**. Співпраця виявилася такою успішною, що до кінця епохи в 2009 році понад 250 мільйонів пристроїв працювало на **Symbian**.

КРОС-ПЛАТФОРМЕНІ ФРЕЙМВОРКИ В СУЧАСНІЙ МОБІЛЬНІЙ РОЗРОБЦІ

- 1. **Xamarin** - це фреймворк для кроссплатформенної розробки мобільних додатків (**iOS, Android, Windows Phone**) з використанням мови **C #**.
- 2. **React Native** - це крос-платформений фреймворк для розробки мобільних додатків. Він був розроблений компанією **Facebook** і випущений у 2015 році.
- 3. **Flutter** — це набір інструментів інтерфейсу користувача (користувальницького інтерфейсу) з відкритим кодом, розроблений **Google** для створення власно скомпільованих мобільних, веб-програм і програм для настільних комп'ютерів з єдиної кодової бази.
- 4. **Apache Cordova** — це відкритий фреймворк для крос-платформеної мобільної розробки. Він дозволяє використовувати стандартні веб-технології для створення мобільних додатків.
- 5. **NativeScript** є фреймворком для крос-платформеної мобільної розробки, який дозволяє використовувати **JavaScript** або **TypeScript** для створення нативних мобільних додатків.

ПЕРЕВАГИ КРОС-ПЛАТФОРМЕНИХ ФРЕЙМВОРКІВ

- **Одноразовий Код**. Розробники можуть використовувати один і той самий код для створення додатків для різних платформ, що зменшує час та зусилля, необхідні для розробки та підтримки додатків.
- **Зниження Витрат**. Використання крос-платформених фреймворків може зменшити витрати на розробку, оновлення та підтримку додатків, оскільки розробники можуть працювати з однією кодовою базою.
- **Швидке розгортання на різних платформах**. Додатки можуть бути швидко розгорнуті на різних платформах, оскільки код може бути перенесений з однієї платформи на іншу
- **Широкий досяг ринку**. Крос-платформені додатки можуть швидше дістатися до ринку, оскільки їх можна розгортати на різних платформах без значних модифікацій.
- **Спільні бібліотеки та компоненти**. Можливість використовувати спільні бібліотеки та компоненти для різних платформ дозволяє прискорити розробку та покращити її якість.

ІНСТРУМЕНТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Компілятор – це програма, яка перетворює вихідний код програми, написаний на високорівневій мові програмування, в еквівалентний йому низькорівневий машинний код або код байткоду.

Автодоповнення – це функція, яка автоматично доповнює код під час його введення. Вона може пропонувати доступні функції, методи, змінні та інші елементи

Система підсвічування синтаксису це функція текстового редактора чи інтегрованого середовища розробки (ICP), яка візуально виділяє різні елементи програмного коду за допомогою кольорів чи інших візуальних ефектів.

ФРЕЙМВОРКИ ДЛЯ РОЗРОБКИ

Django	**Ruby on Rails**	**Spring**
Spring Boot	**Express.js**	**Laravel**

ШТУЧНИЙ ІНТЕЛЕКТ ТА МАШИННЕ НАВЧАННЯ

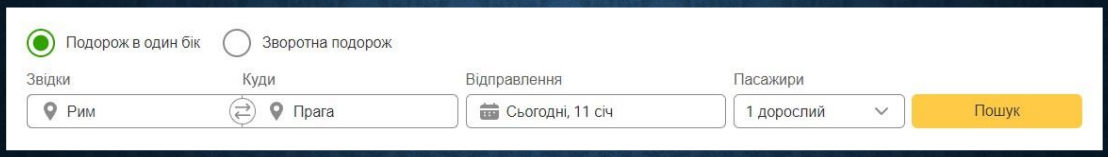
Штучний інтелект — це галузь інформатики, яка займається створенням програм та систем, здатних виконувати завдання, які вимагають інтелектуальної обробки інформації. Основною метою штучного інтелекту є створення комп'ютерних систем, які можуть мислити, вчитися та приймати рішення, схожі на людські.

Машинне навчання – це критично важлива галузь штучного інтелекту, яка привносить революційні зміни в спосіб, яким комп'ютери опрацьовують дані та розв'язують завдання. Застосування машинного навчання вже охоплює широкий спектр сфер, від бізнесу та медицини до науки та техніки.

СПІВПРОГРАМИ (COROUTINE)

- Співпрограма або співпроцедура (англ. **Coroutine**) — компонент програми, що узагальнює поняття підпрограми, додатковою підтримкою безлічі точок входу (а не однієї, як підпрограма) і зупинку та продовження виконання із збереженням певного положення. Співпрограми є гнучкішими і узагальненішими, ніж підпрограми, але рідше використовуються на практиці.

ПОШУК БІЛЕТУ ВІД ТОЧКИ А ДО ТОЧКИ Б



ЯК ПОКРАЩИТИ?

- Оптимізація продуктивності
- Розумний асистент чат-бот
- Розпізнавання обличчя та персоналізований вхід
- Інтерактивність на мапі подорожей
- Екологічна ініціатива

ОПТИМИЗАЦІЯ

- Оптимізація продуктивності важлива для забезпечення ефективності та швидкості роботи вашого сайту бронювання білетів. Деякі поради для оптимізації продуктивності. Запити до бази даних повинні бути оптимізовані можна обмежити кількість великих запитів та використовувати кеш для часто використовуваних запитів
- Використовувати асинхронне завантаження скриптів та ресурсів для того, щоб сторінка завантажувалася швидше.

РОЗУМНИЙ ЧАТ-БОТ

- Додавання розумного чат-бота може значно полегшити взаємодію користувачів із сайтом бронювання білетів.
- Використовувати технології машинного навчання для навчання чат-бота розпізнавати та реагувати на різні запитання користувачів.
- Вдосконалення чат-бота для розпізнавання емоцій у тексті користувача та адаптації відповідей відповідно
- Розробка системи відповідей на найпоширеніші питання для ефективної взаємодії з користувачами

ВИСНОВКИ

Сучасні сайти та додатки для бронювання білетів можуть бути дуже функціональними та зручними для користувачів, але існує кілька аспектів, які можна подолати чи поліпшити для полегшення користування та забезпечення більшого комфорту. Більшість сайтів та додатків пропонують базові функції персоналізації, але є можливість розвивати цей аспект. Персоналізація може включати індивідуальні рекомендації, представлення улюблених напрямків або виділення акцій для конкретного користувача.

Додати можливість використання **VR** або **AR** для попереднього перегляду сидінь у літаку чи поїзді перед бронюванням, щоб користувачі

могли обирати оптимальні місця. Розробити інтеграцію з персональними асистентами

(наприклад, **Google Assistant** чи **Siri**), щоб користувачі могли здійснювати бронювання за допомогою голосових команд.