МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота бакалавра

МОБІЛЬНИЙ ДОДАТОК ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ І ТРЕНУВАННЯ УВАГИ КОРИСТУВАЧА

Здобувач освіти гр. ІНз-81С Руслан КУКСА

Науковий керівник,
кандидат фізико-математичних наук,
асистент кафедри комп'ютерних наук Ольга ШУТИЛЄВА

Завідувач кафедри
доктор технічних наук, професор Анатолій ДОВБИШ

СУМИ 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Затверджую_____
Зав. кафедрою Довбиш А.С.
"_____" _____2022 р.

ЗАВДАННЯ

до кваліфікаційної роботи

здобувача вищої освіти четвертого курсу, групи ІНз-81С спеціальності «122 – Комп'ютерні науки» заочної форми навчання Кукси Руслана Юрійовича.

Тема: «МОБІЛЬНИЙ ДОДАТОК ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ І ТРЕНУВАННЯ УВАГИ КОРИСТУВАЧА»

Затверджена наказом по СумДУ
№_____ від _____2022 р.

Зміст пояснювальної записки: 1) літературний огляд за обраною тематикою роботи; 2) постановка завдання для розробки; 3) вибір оптимальних інструментів для розробки; 4) практична реалізація.

Дата видачі завдання «_____»_____ 2022 р.
Керівник роботи _____Ольга ШУТИЛЄВА
Завдання прийняв до виконання _____ Руслан КУКСА

РЕФЕРАТ

Записка: 65 стор., 26 рис., 1 додаток, 12 джерел.

Об’єкт дослідження – Створення мобільних додатків

Мета роботи – теоретичий та практичний розгляд розробки мобільних додатків для платформи iOS на основі мови програмування – Swift.

Методи дослідження – методи збору та аналізу даних про розробку мобільних додатків.

Результати – готовий мобільний додаток для платформи iOS, що дозволяє користувачу організувати свій день, та який використовує техніку помодоро, що дозволяє користувачу залишатись продуктивним протягом дня.

МОБІЛЬНИЙ ДОДАТОК, SWIFT, XCODE, UI, ФРЕЙМОВОРК, SWIFTUI, IOS

ЗМІСТ

ВСТУП

Актуальність роботи. З кінця 90-х років XX століття, технології все більше і більше вплітаються в повсякденне життя людей. Сьогодні вже майже не можливо знайти людину яка не користується мобільним телефоном або комп'ютером. Усе наше життя, тепер поміщається у маленький пристрій за розміром не більше долоні. Той самий пристрій забезпечує доступ до будь якої інформації у будь якій точці світу у будь який момент часу. Сьогодні ми живемо не тільки у фізичному просторі, а також і у інформаційному. Та не завжди інформація яку ми споживаємо є корисною або сприятливою, як з психологічної точки зору так і з продуктивної.

Перебуваючи у постійному інформаційному потоці де інформація сиплеться на нас з різноманітних джерел нам буває важко зосередитися на своїх основних задачах протягом дня, адже з таким легким доступом до інформації ми постійно перемикаємось між різними інформаційними потоками і відволікаємось на те, що не є життєвою необхідністю для нас, і впадаємо у так звану прокрастинацію, яка виснажує нас як фізично так і психологічно, як наслідок, ми не встигаємо виконати заплановані справи, що пробуджує в нас відчуття провини.

Для подолання прокрастинації та інформаційної залежності сьогодні розроблено багато різноманітних методик. Однією з таких методик є техніка що називається Pomodoro timer. Саме ця техніка буде взята за основу для розробки мобільного додатку, який допоможе користувачам бути більш сфокусованими та продуктивними протягом дня.

Отже **метою кваліфікаційної роботи є** створення мобільного додатку, якій допоможе користувачу бути більш продуктивним і сконцентрованим протягом дня.

1. ЛІТЕРАТУРНИЙ ОГЛЯД

Мобільний додаток, який найчастіше називають додатком, – це тип програмного забезпечення, призначеного для запуску на мобільному пристрої, такому як смартфон або планшетний комп'ютер. Мобільні програми часто служать для надання користувачам послуг, подібних до послуг, доступних на ПК. Програми, як правило, є невеликими окремими програмними блоками з обмеженою функцією. Таке використання програмного забезпечення було спочатку популяризовано Apple Inc. та її App Store, яка пропонує тисячі програм для iPhone, iPad та iPod Touch.

Найпростіші мобільні додатки являють собою портовані програми, що вже існують на базі ПК. Оскільки мобільні додатки стають більш складними, цієї техніки дещо бракує [8]. Більш складний підхід передбачає розробку спеціального мобільного середовища, та використовуючи як його обмеження, так і переваги. Наприклад, додатки, які використовують функції на основі місцезнаходження, за своєю суттю створені з нуля з урахуванням мобільних пристроїв, оскільки користувач не прив'язаний до місця розташування, як на ПК.

1.1 Види мобільних додатків

Додатки поділяються на три великі категорії: нативні, веб-додатки та гібридні.

Нативні програми створені для певної мобільної операційної системи, як правило, iOS або Android. Нативний додаток – це той, який встановлюється безпосередньо на смартфон і може працювати, у більшості випадків, без підключення до Інтернету, залежно від природи програми. Вбудовані програми встановлюються через магазин додатків (наприклад, Google Play або Apple App Store). Зокрема те, що вони розроблені спеціально для однієї платформи і можуть використовувати всі переваги пристрою – вони можуть працювати набагато швидше, використовуючи потужність процесора, і мають

доступ до певного обладнання, наприклад GPS [8]. У деяких смартфонах додаток може керувати пристроями і виконувати роль самого контролера. Вони також можуть включати жести (стандартні жести операційної системи або нові жести, визначені програмою). Нативні додатки можуть використовувати систему сповіщень пристрою.

Рисунок 1.1 – Процес побудови нативних мобільних додатків

Веб-додатки розробляються за допомогою HTML5 і CSS і вимагають мінімум пам’яті пристрою, оскільки вони запускаються через браузер. Користувач перенаправляється на певну веб-сторінку, а вся інформація зберігається в базі даних на сервері. Для використання веб-додатків у користувача зазвичай повинно бути стабільне з’єднання з мережею інтернет.

Гібридний додаток — це додаток, який поєднує елементи як нативних, так і веб-додатків. Гібридні додатки по суті є веб-додатками, які були поміщені в нативну оболонку програми. Після того, як вони завантажені з магазину додатків і встановлені локально, оболонка зможе підключитися до будь-яких можливостей, які надає мобільна платформа через браузер, вбудований у програму. Браузер і його плагіни працюють на сервері і невидимі для кінцевого користувача.

Рисунок 1.2 – Процес побудови гібридних та веб-додактів

Гібридні додатки популярні, оскільки вони дозволяють розробникам писати код для мобільного додатка один раз і при цьому підтримують кілька платформ. Оскільки гібридні програми додають додатковий шар між вихідним кодом і цільовою платформою, вони можуть працювати трохи повільніше, ніж нативні або веб-версії тієї ж програми.

1.2 Основні відмінності між нативними і гібридними мобільними додатками

Як вже було вказано вище, основною відмінністю між нативними і гібридними додатками є те, що вони використовують абсолютно різні технології для побудови, а також те, що нативні додатки розробляються для якоїсь однієї конкретної платформи, у той час як гібридні додатки є мультиплатформенними. Розглянемо також інші відмінності більш детально.

- *Час та вартість розробки*

Гібридні програми є економічно ефективними та займають найменшу кількість часу на розробку. Крім того, гібридні програми легше обслуговувати, оскільки вони мають єдину кодову базу, тоді як рідні програми мають кілька

різних кодових баз, оскільки вони обслуговують спеціально для кожної платформи.

- *Досвід користувача*

Оскільки нативні програми спеціально розроблені для певного магазину додатків, вони забезпечують найкращий досвід користувача. Нативні додатки враховують апаратні можливості та розмір екрана. У порівнянні з гібридними додатками, лише з однією базою коду для всіх платформ, отже, неможливо забезпечити хороший досвід користувача.

- *API*

Нативні додатки можуть мати програмне забезпечення, сумісне з усіма рівнями пристрою, і повні функції керування, такі як GPS, камера та датчики. Гібридні програми підтримують мінімальний рівень SDK, ціле число, що позначає мінімальний рівень API для програми.

Гібридні програми також мають обмежене використання функцій на основі наявності бібліотек сторонніх розробників. Хоча нативні додатки, здається, виграють у цьому, гібридні додатки користуються перевагою єдиного коду для кросплатформного використання.[8]

- *Безпека*

Нативні додатки, як правило, вважаються більш безпечними, ніж гібридні програми. З одного боку, це пов’язано з тим, що нативні додатки мають доступ до вбудованих функцій безпеки для певної платформи. З іншого боку, це також тому, що гібридні програми через використання WebViews можуть бути вразливими до веб-атак. Найпоширеніші атаки на гібридні програми включають введення JavaScript, слабку реалізацію SSL та проблеми з кешуванням. Однак це не означає, що нативні додатки невразливі.

- *Підтримка та обслуговування*

Нативні додатки користуються повною підтримкою в App Store і Play Store. Проте гібридні програми покладаються на сторонніх розробників для розгортання оболонки програми. Обгортка – це WebView, створений із

двійкових файлів. Це дозволяє програмі взаємодіяти з платформою пристрою та включати функції операційної системи.

Гібридні програми отримують підтримку від оболонки, яка надає виправлення для виправлення програм. Нативні програми пропонують більш високу підтримку в автономному режимі порівняно з гібридними програмами, оскільки їх статична інформація не зберігається на сервері [9].

App Features	Native App	VS	Hybrid App
Programming Language	Native Only		Html, css, javascript
Platform	Single Platform		Multiple Platform
Performance	Faster and more reliable		Slower and less reliable
Speed	High		Medium
Developing Time	More		Less
Customer Satisfaction	Fully		Based on app Performance
User Interface	Functionality rich and more attractive		Never give user a fully native experience
Codebase	Separate for native platform		Single For All Platform
Access to Device-Specific Features	All		Limited
Access to Mobile Devices Hardware	High		Moderate
Access to Native Api's	High		Moderate
Development Cost	Expensive		Affordable

Рисунок 1.3 – Порівняльна таблиця нативних і гібридних мобільних додатків

- *Масштабованість*

Нативні додатки ефективно використовують складні функції. Ці функції бездоганно взаємодіють з операційною системою та обладнанням. Гібридні програми стають важкими та повільнішими з додаванням складних компонентів. Їх потрібно розширити для різних операційних систем, оскільки вони використовують веб-технології, а не нативні програми.

1.3 Хід розробки мобільних додатків

Індустрія мобільних додатків також розвивалася разом із розвиненими технологіями. Від сектору фінансових технологій та страхування до індустрії моди та роздрібної торгівлі – розробка мобільних додатків є необхідною для всіх. Те, що починалося з такого простого, як додаток Flappy Bird, тепер перейшло до того часу, коли вони задовольняють всім потребам людини [9].

Хоча запити користувачів та інноваційність бізнесу разом перетворили мобільні додатки, процес розробки залишається незмінним лише з незначними змінами, будь то доповнення чи видалення процесу.

Індустрія розробки мобільних додатків, хоча й дуже обширна з точки зору гравців, дотримується приблизно такого ж процесу, коли справа доходить до розробки комп'ютерних програм. Ось як виглядає узагальнений процес розробки мобільного додатка:

Рисунок 1.4 – Життєвий цикл розробки мобільних додатків

Як видно на інфографіці про життєвий цикл розробки мобільних додатків, етапи ефективної розробки програмного продукту складаються з трьох частин – підготовчих елементів, фактичної стратегії/процесу розробки мобільних додатків і рекомендацій з обслуговування після запуску для ефективного процесу розробки програмного продукту, що є досить стандартним процесом при розробці мобільних додатків. Оскільки кожна програма відрізняється, тому методології розвиваються відповідно до вимог.

Ці етапи розробки мають додаткові процеси, які зазвичай включають ідею, стратегію, процес розробки мобільного додатка, розробку, розгортання та етапи після запуску.

1.4 Підготовчий процес або етап дослідження

Серед основних пунктів даного процесу, можна виділити наступні:

- Збирання та аналіз бізнес вимог
- Наявність чіткого бачення про цілі і призначення мобільного додатку
- Розуміння функціональних вимог до мобільного додатку
- Розробка дорожньої карти

Як випливає з назви, це етап, на якому підприємства займаються дослідженнями. Метою на цьому етапі є встановлення життєздатності ідеї програми. Щоб досягти цього, компанії збирають глибоке розуміння проблеми, яка прагне вирішити демографічні показники користувачів, і розуміння конкурентів, які пропонують подібне рішення або частину рішення, яке збирається запропонувати додаток.

Після визначення ідей фактично виникає необхідність розробити стратегію щодо процесу, наприклад, як це зробити, з чого почати та які речі потрібні.

1.5 Робочі процеси розробки мобільного додатку

Після завершення та формування бізнес-вимог починається фактичний процес – розробка. Процес розробки мобільних додатків, складається з двох частин – це Front-End розробка та Back-End розробка.

В свою чергу, Front-End розробка складається з двох окремих етапів. Це розробка користувацького інтерфейсу, та досвіду користувача (UI/UX), а також розробка безпосереднього самого мобільного додатку.

Мета полягає в тому, щоб запропонувати простий досвід із дуже меншою кривою засвоєння. Крім того, він повинен бути ідеально синхронізований із бекендом, щоб легко обробляти інформацію.

- *Front-End Development*

Видима частина веб або мобільного додатка – це частина, з якою користувач взаємодіє безпосередньо. Зазвичай її називають «клієнтською стороною» програми. Інтерфейс складається з усього, що користувач бачить під час взаємодії з веб-сайтом або додатком, наприклад кольори та стилі тексту, фотографії, графіки та таблиці, кнопки, кольори, меню навігації та багато іншого. Розробники інтерфейсу забезпечують структуру, зовнішній вигляд, поведінку та вміст усього, що відображається на дисплеях браузера під час відкриття веб-сайтів, онлайн-додатків або мобільних додатків. Ключові моменти розробки інтерфейсу – це реагування та продуктивність. Розробник інтерфейсу повинен переконатися, що сайт адаптивний, тобто він правильно працює на пристроях будь-якого розміру. Продуктивність програми повинна бути стабільною в будь-який час, незалежно від пристрою, який використовується для доступу до програми [7].

- *Back-End Development*

Бекенд також відомий як серверна сторона веб-сайту або програми. Він організовує та зберігає дані, а також забезпечує належне функціонування всього на клієнтській стороні веб-сайту. Хоча бекенд не взаємодіє з користувачами безпосередньо, він відіграє незамінну роль за лаштунками, додаючи ключові функції. Без добре написаного бекенда інтерфейс додатку не працюватиме належним чином. Таким чином, навіть якщо безпосередньо немає взаємодії з бекендом, як з інтерфейсом, то опосередковано користувач контактує з усіма процесами, що відбуваються на серверній частині.

Бекенд включає в себе такі дії, як написання API, створення бібліотек і робота з системними компонентами без інтерфейсу користувача. Основною функціональністю додатків зазвичай керує бекенд. Наприклад, якщо користувач хочете щось придбати в інтернет-магазині, бекенд керує фактичними грошовими трансакціями під час проходження процесу оплати. У той час як інтерфейс перевіряє, чи кнопка «оформити» розміщена належним чином на сторінці та спрямовує вас на наступну сторінку, бекенд взаємодіє зі

службами за межами додатка чи веб-сайту, як наприклад банківський додаток або PayPal. Ми не бачимо, як відбувається весь процес цієї транзакції, оскільки все відбувається за лаштунками, у бекенді.

- *MVP*

Мінімально життєздатний продукт (MVP) – це версія продукту з достатньою кількістю функцій, щоб їх могли використовувати ранні клієнти, які потім можуть надати зворотній зв'язок для майбутньої розробки продукту.

Зосередженість на випуску MVP означає, що розробники потенційно уникають тривалої та (зрештою) непотрібної роботи. Натомість вони повторюють робочі версії та відповідають на відгуки, оскаржуючи та підтверджуючи припущення щодо вимог продукту. Термін був придуманий і визначений у 2001 році Френком Робінсоном, а потім популяризований Стівом Бланком та Еріком Райсом. Це також може передбачати проведення попереднього аналізу ринку. MVP аналогічний експерименту в науковому методі, який застосовується в контексті перевірки бізнес-гіпотез. Він використовується для того, щоб майбутні підприємці знали, чи буде дана бізнес-ідея насправді життєздатною та прибутковою, шляхом перевірки припущень, що лежать в основі продукту чи бізнес-ідеї. Концепція може бути використана для підтвердження ринкової потреби в продукті і для поступового розвитку існуючого продукту [10]. Оскільки він перевіряє потенційну бізнес-модель для клієнтів, щоб побачити, як відреагує ринок, він особливо корисний для нових/початківців компаній, які більше стурбовані з'ясуванням потенційних можливостей для бізнесу, а не реалізацією готової ізольованої бізнес-моделі.

1.6 Підтримка та обслуговування після запуску

Підтримка після запуску програми – це діяльність, яка вимагає такого ж часу та зусиль, як і фактичний процес розробки. Тому тестування вимагає часу. Запуск процесу перевірки коду за допомогою повного тестування

забезпечення якості (QA) на етапі розробки допомагає зробити додаток безпечним, стабільним і придатним для використання, а також гарантує, що команді не залишиться жодних серйозних помилок. Для комплексного тестування QA програми вам спочатку потрібно спланувати тестові випадки, які охоплюють усі аспекти тестування програми. Після того, як QA повністю завершено, настає процес запуску.

Існує два способи розгортання додатків. Перший передбачає запуск вашого веб-сервера (API) у масштабованому виробничому середовищі. Другий включає запуск програми в Google Play Store і Apple App Store.

На цьому процес не закінчується, все ще триває обслуговування та оновлення програми відповідно до зручності та потреб користувачів.

Компанія яка виступає розробником, повинна переконатися, що їх додаток був прийнятий і сподобалась користувачам у магазинах додатків. І це буде гарантовано лише тоді, коли компанія що розробляє додаток буде стежити за тим, як користувачі реагують на це – якщо вони просять оновити чи виправити помилки тощо [8].

1.7 Інтерфейс користувача та UX-дизайн

Що перше, що бачать користувачі, відкриваючи додаток? Це не функції, які він пропонує, а те, як він виглядає і як він взаємодіє з користувачами. Хоча ключовим елементом мобільного додатку є проблема, яку він вирішує, легке та зручне керування його функціями робить мобільний додаток привабливим для ваших клієнтів [7]. Функціональність програми марна, якщо користувачі не можуть реально реалізувати її в реальному житті, і вони або виберуть інший мобільний додаток, або зашкодять репутації ненависними відгуками. Зазвичай вони роблять обидва.

Інтерфейс користувача (UI) відноситься до того, як мобільний додаток представляє себе користувачеві. Досвід користувача (UX) стосується того, як користувач взаємодіє з додатком і дає відповідні команди для завершення або

вирішення проблем, для яких спочатку було розроблено програму. Ефективність додатку можна оцінити за оптимальним поєднанням його функціональності та привабливості. Зважаючи на це, треба бути переконаним, що додаток відповідає 4 основним правилам:

• Він інтуїтивно зрозумілий

• Він простий у використанні

• Додаток захоплює користувача

• Додаток виконує ті задачі які перед ним поставлені

Впроваджуючи модний дизайн мобільного додатка, треба бути переконаним, що він не тільки виглядає красиво, але й взаємодіє з користувачем і робить початкове використання швидким, зручним і вигідним.

1.7.1 Google Material Design проти iOS Human Interface Design

Сьогодні процес розробки мобільних додатків передбачає розробку для двох найпопулярніших мобільних операційних систем: Android та iOS. Звичайно, ви можете зосередитися на тій, яка, на думку компанії, буде кращою платформою для послуг, які вона пропонуєте, але слід пам’ятати, що також потрібно враховувати основну мету мобільного додатка.

Для належного охоплення потенційної аудиторії найкращим варіантом є розробка як Android, так і iOS-версій додатка одночасно. Хоча вони будуть одним і тим же додатком і матимуть однакові цілі, різні операційні системи та дизайн пропонують різні підходи [7]. Копіювання та перенесення одного і того ж дизайну, швидше за все, не принесе успіху.

Human Interface Guidelines (HIG) – це документи з розробки програмного забезпечення, що були розроблені компанією Apple, і які пропонують розробникам додатків набір рекомендацій. Їхня мета – покращити досвід для користувачів, роблячи інтерфейси програм більш інтуїтивно зрозумілими, зрозумілими та послідовними. Більшість посібників обмежуються визначенням загального вигляду програм у певному середовищі (iOS, iPadOS, MacOS). У посібниках перераховано конкретні правила. Політика цих

рекомендацій іноді ґрунтується на дослідженнях взаємодії людини та комп'ютера (так звані дослідження юзабіліті), але більшість базується на конвенціях, обраних уподобаннями розробників платформи.

Основна мета HIG – створити узгоджений досвід у всьому середовищі включаючи програми та інші інструменти, що використовуються. Це означає як застосування однакового візуального дизайну, так і створення послідовного доступу та поведінки загальних елементів інтерфейсу – від простих, таких як кнопки та значки, до більш складних конструкцій, таких як діалогові вікна.

Material Design під кодовою назвою Quantum Paper був випущений у 2014 році як мова дизайну. Цей, як і будь-який інший дизайн, має набір вказівок і принципів для дотримання правил дизайну матеріалів. Кожен, хто прагне зробити свій Android додаток з використанням Material Design, повинен дотримуватися цих правил, щоб надати однаковий вигляд на всіх пристроях Android.

Справжньою ідеєю розробки цієї мови було введення інтерактивного інтерфейсу. Користувальницький інтерфейс Apple з плоским дизайном менш інтуїтивно зрозумілий, тоді як користувачів бентежить велика площинність значків. Це означає, що в Material Design кнопки, які можна натиснути, змішуються з текстами та значками, які не можна натискати. Для вирішення цієї проблеми, зокрема, матеріальний дизайн Google працює з деякими скевоморфними та неоморфними елементами з простим дизайном [8].

Незважаючи на те, що Material Design виглядає плоским, він багатовимірний, для тіней і спливаючих ефектів. Простіше кажучи, матеріальний дизайн – це вдосконалена форма інтерфейсу плоского дизайну, яка надає пріоритет маленьким деталям в анімації, використання різних відтінків і шарів, щоб виділити об'єкт. Material Design допомагає користувачеві плавно переміщатися по додатку, зберігаючи чистий інтерфейс користувача.

Material Design Google пропонує розробникам створювати програми з однаковими іконками на різних платформах, та сприймається всіма розробниками, оскільки значки ще більш виразні і, безсумнівно, допомагає користувачеві орієнтуватися.

І Google, і Apple змагаються, щоб надати своїм користувачам інтуїтивно зрозумілий інтерфейс користувача. Мета полягає в тому, щоб створити такий інтерфейс, за допомогою якого люди не заплутаються і зможуть користуватися додатками, веб-порталами, пристроями з нульовими труднощами. За загальним визнанням, підхід, використаний Google, є інтуїтивним, але також інтерактивним. Анімації, які використовуються в інтерфейсі матеріального дизайну, піддаються цілеспрямованій діяльності. Тоді як інтерфейс користувача Apple Flat Design, який тепер називається дизайном людського інтерфейсу, є плоским, чистим і зручним для користувачів.

1.7.2 Каркасний дизайн

Це та частина, коли ідея стає невеликим проектом. Це ще не мобільний додаток, а скоріше ескіз майбутнього мобільного додатка. У сфері розробки додатків це називається Wireframe.

Каркас (Wireframe) – це спрощений базовий план майбутнього проекту, який має на меті

- Візуально представляти мобільний додаток
- Показувати, як мобільний додаток повинен працювати
- Описати його основні особливості
- Показати його внутрішню навігацію

Каркаси не обов’язково повинні бути красивими і прискіпливими до деталей. Можна вважати, що каркас є тестовим полем для експериментів, де можна випробовувати свої ідеї, ділитися своїми думками та порівнювати можливі взаємодії тощо [7].

Wireframes – це як цифрові ескізи, розроблені дизайнерами для процесу розробки мобільних додатків. Це допомагає представити концептуальні макети програми, також відомі як макети низької точності, щоб визначити візуальну структуру відповідно до функціональних вимог вашої програми [9].

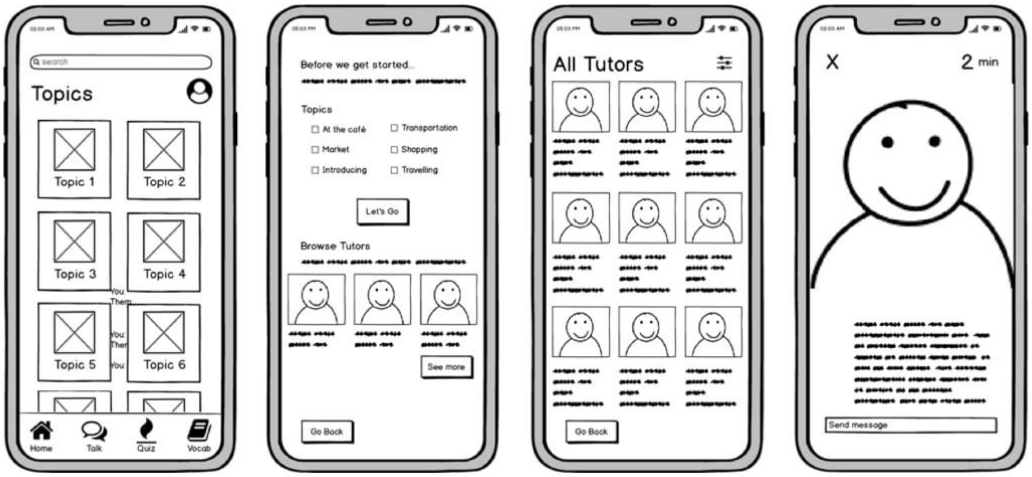


Рисунок 1.5 – Приклад каркасного дизайну (Wireframes)

Wireframes більше зосереджуються на естетиці додатку та розробці мобільних додатків для його користувацького досвіду, а не на колірних схемах і стилях програми. Це економічно ефективний і швидкий підхід, який повинен враховувати дизайн для конкретного пристрою на широкому діапазоні пристроїв, щоб забезпечити інтуїтивно зрозумілий і специфічний для користувача досвід.

1.7.3 Макети

Це остаточний дизайн мобільного додатка, який також відомий як високоякісний дизайн у сфері розробки мобільних додатків. Можна створювати макети, оснащуючи каркас додатка своїм посібником зі стилю. Після завершення дизайну програми ваша інформаційна архітектура, робочий процес та естетика будуть змінені пізніше. Для створення високоякісних макетів в процесі розробки мобільних додатків можна використовувати такі інструменти як Adobe Photoshop, Illustrator, Sketch тощо [8].

1.7.4 Прототипи

Макети в розробці програм – це статичні проекти, які можуть представляти функціональність мобільного додатка. Його можна перетворити на прототипи, які є інтерактивними, тобто можна побачити як працює

навігація у додатку або що відбувається при натисканні на той чи інший елемент на екрані. Прототипи важливі для імітації користувацького досвіду та робочого процесу програми, що очікується від готового продукту. Хоча розробка прототипів може зайняти час, зусилля варті того, тому що вони демонструють дизайн і функціонування додатка на ранній стадії. Прототипи часто допомагають у процесі розробки мобільного додатка, щоб визначити зміни у функціональності, запропоновані на ранньому етапі розробки.

Зокрема, коли функціональні сторони програми недостатньо продумані, деякі фірми вирішують розробляти прототипи на етапі каркасного створення. Вони також роблять це, щоб переглянути запропоновані функції програми з фокус-групою під час розробки мобільних додатків.

1.8 Організація процесів розробки мобільних додатків

Коли розробка мобільного додатка вже пройшла етапи створення каркаса та прототипу, настав час поглянути на процес організації. Робочий процес додатку слід сприймати як створення розширених списків справ для всієї родини з конкретними завданнями, напрямками та часовими рамками. Без належної організації етапів розробки додатку, практично неможливо успішно розпочати процес розробки та підтримувати його в правильному напрямку.

Програми робочого процесу дозволяють проактивно контролювати та керувати командою розробників, призначати завдання, вносити зміни на ходу та гарантувати, що кожен член команди постійно володіє останньою інформацією. Хоча все це можна зробити за допомогою електронної пошти та телефонних дзвінків, наявність планування в реальному часі змінює гру [8].

Беручи участь у процесі розробки додатків, контроль бюджету є ключем до успішного результату. Добре організований та автоматизований робочий процес додатків допоможе обмежити час, необхідний для затвердження бюджету, встановити пріоритетні витрати та регулювати непотрібні витрати.

Зрештою, можна побачити статистику по кожному відділу та реорганізувати грошовий потік (за потреби).

Управління командою розробників – процес завжди непередбачуваний, і навіть з найкращими розрахунками він, ймовірно, потребує кращих ресурсів, будь то грошовий дохід або штат співробітників. Можливість керувати, контролювати та стежити за етапами розробки мобільних додатків дозволяє миттєво реагувати на надзвичайні ситуації, перенаправляти грошовий потік у необхідні відділи та зменшувати або збільшувати кількість співробітників для досягнення оптимального процесу розробки додатків.

Маркетинговий план. Маркетинг майбутнього мобільного додатка настільки ж важливий, як і його розробка. Правильний маркетинговий менеджмент дозволяє привернути увагу потенційних інвесторів, краще запускати краудфандингові кампанії (Kickstarter, IndieGoGo тощо) і активізувати інтерес у майбутніх клієнтів. Добре організований робочий процес програми дає зворотний зв’язок у реальному часі від людей, реалістичні очікування від інвесторів і привертає увагу вашого мобільного додатка. Робочий процес із належним підходом може зробити процес розробки набагато плавнішим.

1.9 Архітектура мобільних додатків

Створюючи програму, архітектори повинні мати план, перш ніж почати. Архітектура програми – це базова структура, з якою розробники працюють, щоб створити додаток. Ця основа описує інструменти та методи, які необхідно використовувати.

При створенні мобільного або веб-додатка важливо переконатися, що кожен компонент виконує свою функцію. Невелика проблема на етапі створення архітектури для мобільного додатка може мати далекосяжні наслідки для життєздатності кінцевого продукту [7]. Популярні мобільні додатки розуміють цей принцип – вони створені з урахуванням стабільності та функціональності та

щодня демонструють ці концепції. Додатки, які не враховують архітектуру розробки мобільних додатків у своєму плануванні, як правило, не живуть довго в жодному з магазинів додатків, оскільки користувачі швидко демонструють своє невдоволення. Архітектура безпеки мобільних додатків є ще однією ключовою особливістю надійного додатка, незалежно від iOS, Android або гібридних додатків. Вибираючи правильну архітектуру та розробляючи додаток, потрібно намагатися уникати лазівок у безпеці мобільних додатків [7].

Обговорюючи архітектуру розробки мобільних додатків, архітектори звертають увагу на кращі практики та стандарти в галузі. У цьому аналізі розробникам потрібно враховувати всі типи бездротових продуктів, щоб програма безперебійно працювала як зі смартфонами, так і з планшетами.

Хоча цей рівень інформації підходить для пояснення високого рівня, його можна деталізувати, щоб отримати ще більше розуміння базової архітектури мобільних додатків. У цьому випадку потрібно зрозуміти цільову аудиторію та те, як вона буде використовувати мобільний додаток.

Інші міркування, які слід пам’ятати, – це пристрої, які потрібно підтримувати вашим мобільним додатком. Це включає розуміння специфікацій обладнання, таких як розмір екрана та роздільна здатність, а також базову потужність процесора та пам’ять. Залежно від варіанту використання для правильної роботи додатку може знадобитися камера або GPS, що вплине на прийняття рішень. У будь-якому випадку дизайн має базуватися на мінімальних вимогах проти ідеальних рішень. Слід також враховувати те, що у користувачів може бути обмежене підключення до мережі інтернет, оскільки не завжди можна бути впевненим, як кінцеві користувачі отримають доступ до мобільного додатка та як його використовуватимуть у польових умовах.

Ще одна важлива функція, яку слід враховувати при виборі шаблону архітектури, – це безпека програми. Безпека мобільних додатків – це захід для захисту додатка від зовнішніх загроз, таких як зловмисне програмне

забезпечення та інші цифрові шахрайства, які викривають важливу особисту, корпоративну та фінансову інформацію.

Оскільки лазівки в мобільній безпеці можуть надавати доступ до конфіденційної інформації, а також розкривати такі дані, як поточне місцезнаходження користувачів, банківська інформація, особиста інформація тощо, важливо включати функції безпеки в архітектуру програми.

1.9.1 Архітектурні рівні

З простої точки зору, програми розроблені з урахуванням трьох різних шарів. Розглянемо їх більш детально.

- *Рівень презентації*

Рівень презентації звертає увагу на компоненти інтерфейсу користувача та компонентів процесу UI. Основний фокус цього рівня полягає в тому, як програма буде представлена кінцевому користувачеві. Під час розробки цього шару розробники повинні визначити тип клієнта, який відповідає інфраструктурі.

Рівень презентації охоплює компоненти UI та компоненти процесу UI. Обговорюючи цей шар, розробник додатка повинен визначити, як мобільний додаток буде представляти себе перед кінцевим користувачем. На цьому етапі також слід визначити важливі речі, такі як теми, шрифти, кольори тощо.

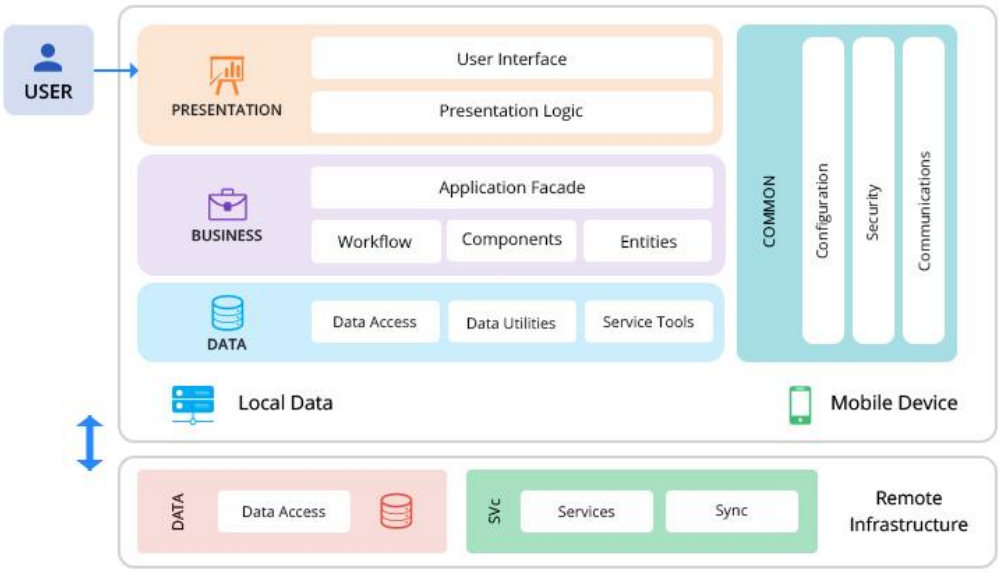


Рисунок 1.6 – Архітектурні рівні

- *Бізнес-рівень*

Цей рівень являє собою ядро мобільного додатка, яке розкриває функціональні можливості. Рівень бізнес-логіки може бути розгорнутий на сервері до якого користувачі отримують доступ віддалено за допомогою мобільного додатка, щоб зменшити навантаження. Таке навантаження пов’язано з обмеженими ресурсами, доступними на мобільних пристроях. Рівень бізнес-логіки включає в себе робочі процеси, бізнес-компоненти та сутності під капотом.

- *Рівень доступу до даних*

Цей рівень створюється з комбінації утиліт даних, компонентів доступу до даних і агентів обслуговування. Рівень доступу до даних відповідає вимогам додатків і забезпечує безпечні транзакції даних. Важливо спроектувати цей шар, оскільки він може масштабуватися в майбутньому. Це дані третього етапу, оскільки вони включають компоненти доступу до даних, помічники/служби даних та агенти обслуговування.

Іншим фактором для розробки цього шару є вибір правильного формату даних і застосування надійної техніки перевірки. Таким чином, ваш додаток можна захистити від недійсного введення даних. Розробники мобільних додатків повинні зосередитися на відокремленні бізнес-логіки від коду рівня презентації.

1.9.2 Принципи чистої та легкої в обслуговуванні архітектури

Хороша архітектура мобільного додатка гарантує, що компоненти мають кілька рівнів відповідальності. Або хороша архітектура мобільного додатка – це та, яка забезпечить виконання принципів і хороших моделей програмування, таких як SOLID або KISS. Серед найбільш важливих принципів можна виділити наступні:

- *Розділення обов’язків*

Цей принцип означає, що різні сутності повинні бути розділені в коді. Це допомагає повторно використовувати ці сутності, систематизувати

розробку, легко налагоджувати та ізолювати частини, що часто змінюються, щоб вони не впливали на інші.

- *Тестувальність*

З відповідною архітектурою це набагато простіше. Команді тестувальників буде корисним писати тестові випадки та мати можливість тестувати функціональні можливості окремо. Ви уникнете пошуку проблем під час виконання, що зазвичай означає виправлення на тиждень.

- *Надійність*

Добре підібрана архітектура має вирішальне значення для створення стабільних програм без серйозних невідповідностей, оскільки вона визначає, як частини коду взаємодіють один з одним.

- *Масштабованість*

Програма повинна мати міцну основу для створення нових функцій і змін відповідно до потреб бізнесу. Він також має бути придатним для майбутніх оновлень (наприклад, нових бібліотек, операційних систем) у технологіях програмування.

- *Обслуговування і простота використання*

Хороша архітектура спрощує код як для написання, так і для читання. Це також може призвести до низької вартості обслуговування. Виконання всіх цих умов дозволяє прискорити розвиток і значно полегшити подальше обслуговування. Таким чином, це заощаджує час і гроші. Проте, мудро підібрана архітектура разом із технологіями, що відповідають платформі, як-от Swift для iOS або Kotlin для Android, будуть найкращими для вирішення складних бізнес-проблем найефективнішим способом для мобільних проектів.

Це дозволить уникнути багатьох проблем, які виникають у випадку використання гібридних технологій, що в свою чергу дозволить заощадити час і гроші в довгостроковій перспективі. Хороша архітектура повинна бути настільки абстрактною, щоб її можна було застосувати до таких платформ, як iOS або Android. Однією з найважливіших характеристик гарної архітектури є поділ рівня відповідальності.

1.9.3 Види архітектурних шаблонів

Шаблони дизайну зробили величезний вплив на розробку програмного забезпечення. Подібно до веб-додатків, впровадження мобільних додатків також встановило деякі перевірені моделі та стандарти для подолання проблем і обмежень у розробці мобільних додатків. Більшість мобільних додатків було розроблено з низькоякісним кодом і вони не засновані на архітектурних шаблонах проектування. Розробка мобільного додатка з правильним шаблоном дизайну може ефективно зв'язати інтерфейс користувача з моделями даних і бізнес-логікою. Це вплине на те, як виглядатиме ваш вихідний код. Існує досить багато шаблонів архітектурного проектування для мобільної розробки. Розглянемо найпоширеніші з них [9].

- *Classic Model-View-Controller (MVC)*

MVC – це перший підхід до опису та реалізації розробки програмного забезпечення на основі їхніх обов'язків. Трюгве Рінскауг представив архітектуру MVC у Smalltalk-76 у 1970-х роках. В основному розроблений для настільних комп'ютерів, він широко використовується як архітектура веб-програм основними мовами програмування. MVC включає в себе три компоненти для кожного об'єкта, а саме:

- Модель – відповідає за дані або рівень доступу до даних.
- Представлення (View) – відповідає за графічне відображення даних.
- Контролер – він служить сполучною ланкою між представленням і моделлю. Контролер не відіграє ролі посередника між представленням і моделлю. Він змінює модель на основі активності користувача на View, а також оновлює зміни за допомогою View.

Як показано на рисунку 1.7 – View може отримати прямий доступ до моделі, але в режимі лише для читання – перегляд не повинен змінювати стан моделі. Контролер може це зробити; однак контролер не реагує на View безпосередньо. Відповідальність View – запитувати статус і отримувати статус зміненої моделі [11].

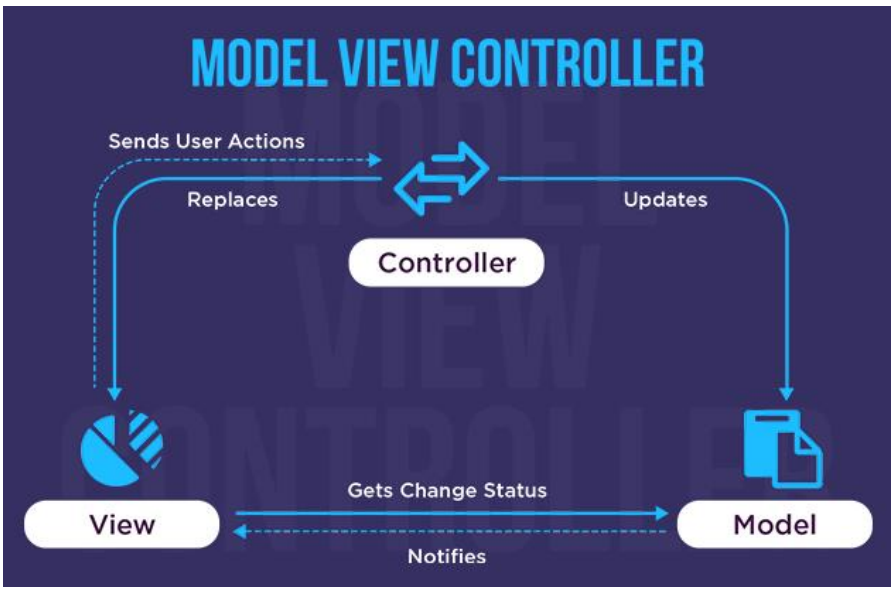


Рисунок 1.7 – Принцип роботи Classic Model-View-Controller

Контролер не несе відповідальності за цю передачу між представленням і моделлю. Діаграма послідовності полегшує розуміння взаємодії між представленням, моделлю та контролером:

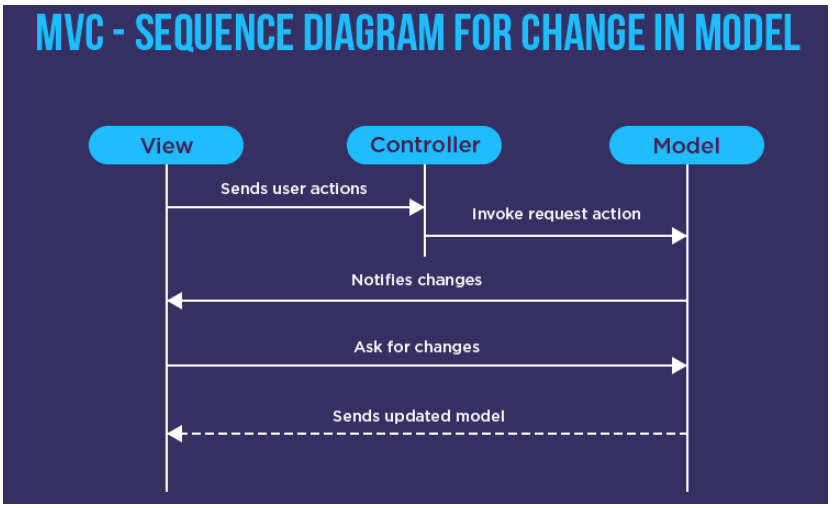


Рисунок 1.8 – Діаграма послідовності змін в Classic MVC

Контролер з’являється лише тоді, коли потрібно відтворити нове представлення. Наприклад, якщо ви просто хочете оновити View, щоб відобразити дані в іншому форматі, контролер може зробити це, не вимагаючи оновлення в моделі. Недоліки шаблону проектування MVC – усі три

компоненти тісно пов’язані, і це різко впливає на можливість повторного використання кожного компонента, оскільки кожен компонент знає про два інших. Отже, класичний MVC не є кращим для розробки сучасних мобільних додатків [11].

- *Apple MVC або розширена архітектура MVC*

Розширений MVC є результатом спроби адаптувати класичний MVC у мобільній розробці. У цьому MVC немає прямого зв’язку між представленням і моделлю, а контролер служить посередником між ними.

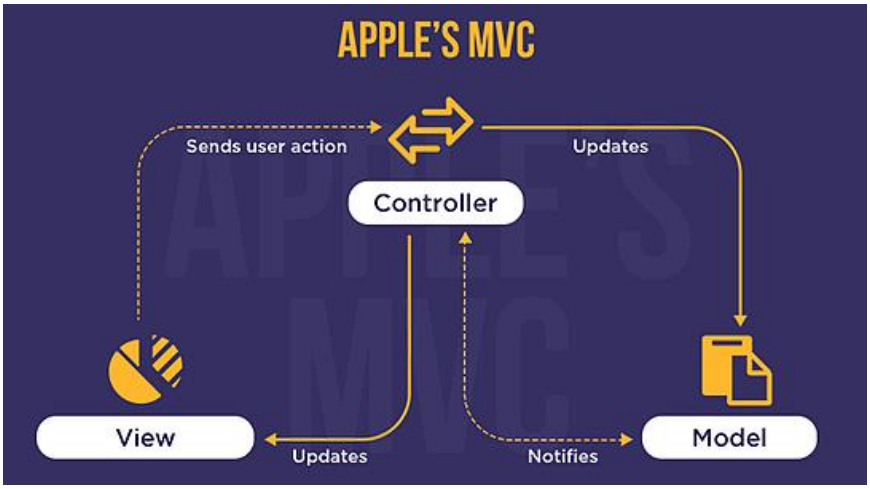


Рисунок 1.9 – Принцип роботи Apple MVC

Контролер надсилає повідомлення до моделі на основі вхідного запиту, викликаного представленням. Модель повідомляє про зміни стану контролеру, а контролер оновлює зміни в об’єкті View. Зв'язок між моделлю та представленням здійснюється непрямим шляхом через контролер. Представлення не турбується про те, як виконуються дії, але підтримує подання графічного інтерфейсу та делегує контролеру специфічні для програми рішення щодо поведінки інтерфейсу.

Клас контролера реалізує інтерфейс контролера програми, який виконує дії на основі запиту користувача, наприклад збереження даних, введення даних, скасування дії тощо. Діаграма послідовності пояснює передачі в архітектурі Apple MVC [11]:

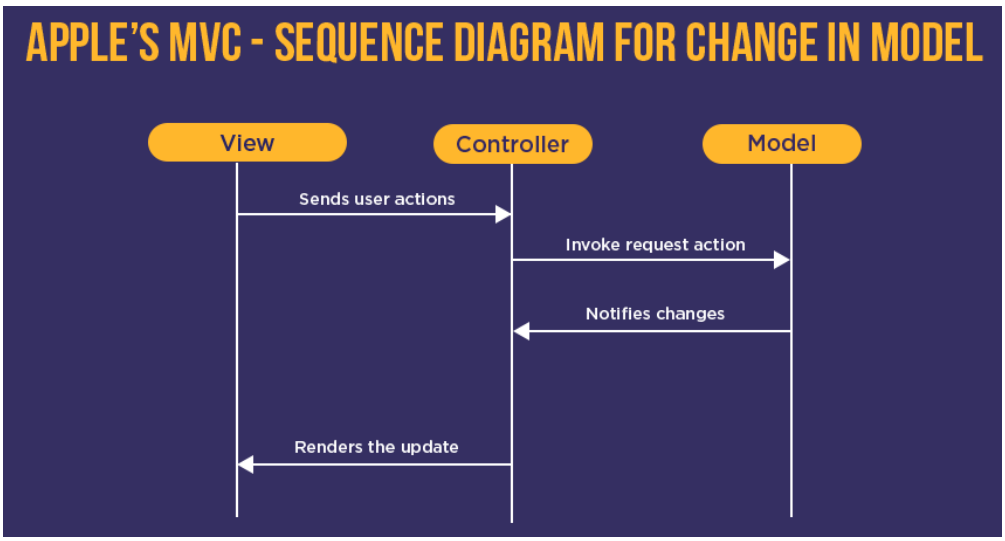


Рисунок 1.10 – Діаграма послідовності змін в Apple MVC

У цьому шаблоні проектування немає прямого зв'язку між моделлю та представленням. Крім того, контролер включає в себе логіку обробки представлення. Такий розподіл обов'язків більше підходить для розробки додатків, але оскільки контролер бере участь у життєвому циклі View, важко позначити, що вони розділені.

Хоча можна розвантажити частину бізнес-логіки View з моделі, є менше шансів ініціювати розвантаження, оскільки більшість часу View зайнятий надсиланням дій користувача до контролера. Крім того, контролер відповідає за все, і, отже, розмір контролера збільшується. Крім того, коли View тісно пов'язаний з контролером, проводити модульне тестування стає важче [9].

- *Model View Presenter (MVP)*

Третя ітерація, яку слід розглянути для розробки мобільних додатків, – це MVP (Model View Presenter), який розроблено на основі MVC і широко використовується в розробці веб-додатків з 1990 року. Можете реалізувати MVP як варіанти пасивного представлення (Passive View), моделі презентації (Model Presentation) або супервайзера контроллера (Supervasing Controller). Наступна архітектура зосереджена на варіанті Passive View, який є більш вигідним, ніж дві інші альтернативи.

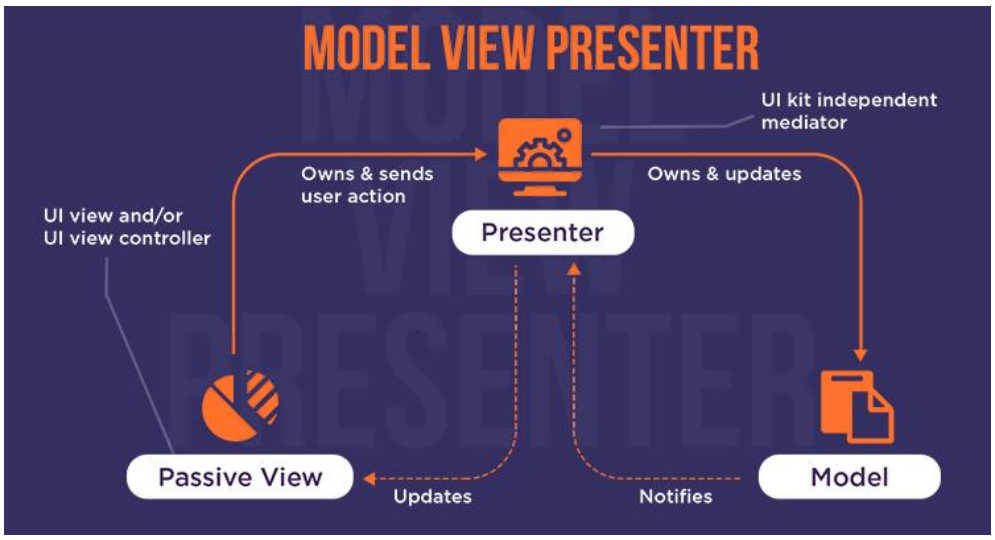


Рисунок 1.11 – Принцип роботи Model View Presenter

Може здатися, що це схоже MVC від Apple, але це не так. Тут Presenter служить посередником між пасивним представленням і моделлю. На відміну від Apple MVC, Presenter і Passive View не тісно пов'язані один з одним. Представлення стає пасивним і не несе відповідальності за оновлення на основі змін у моделі. Presenter оновлює дані та стан представлення [11].

Він вирішує вищезазначені проблеми з шаблонами MVC. Крім того, такі підкласи як UIViews і UIViewController присутні в Passive View, а не в Presenter. Розглянемо послідовність комунікації в MVP:

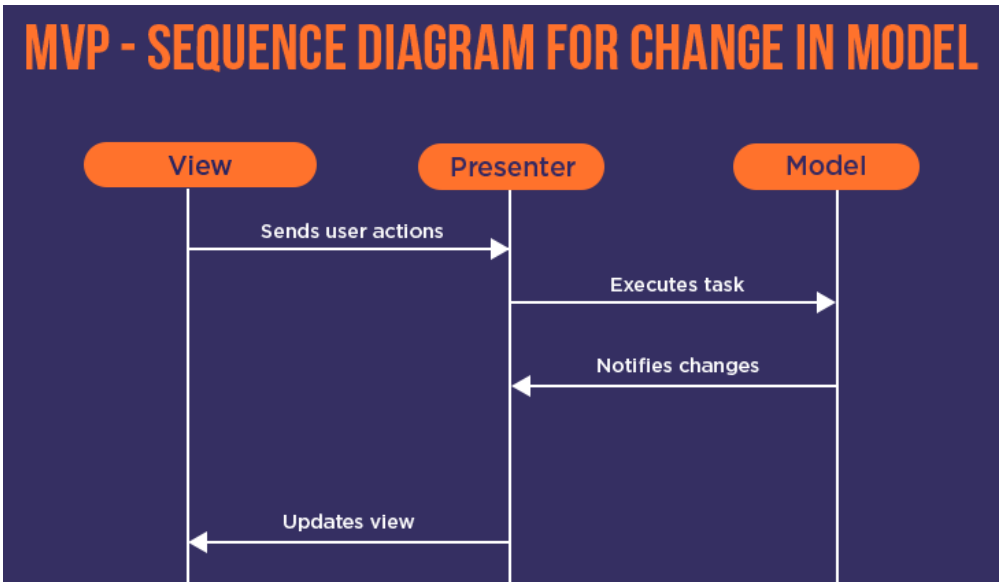


Рисунок 1.12 – Діаграма послідовності змін в MVP

Переваги перед шаблоном проектування MVC:

- Немає прямого зв’язку між моделлю та пасивним представленням
- Презентер не бере участь у життєвому циклі View
- Пасивне представлення не знає про існування моделі та презентера, а пасивність покращує можливість тестування.

Недоліки шаблону проектування MVP:

- Виникає проблема зі збиранням через наявність трьох окремих слоїв.
- Складно реалізувати логіку зв’язування без прямого доступу до елементів керування інтерфейсу користувача. Залежність від кожного представлення, щоб реалізувати інтерфейс, а також дозволити презентеру спілкуватися з представленням.
- Це вимагає виконувати всі роботи зі зв’язуванням вручну, отже, він не підходить для автоматизованої розробки додатків.

▪ *Архітектурний шаблон Model-View-ViewModel (MVVM)*

Наступною ітерацією цього підходу, яка вирішує вищезазначену проблему, є MVVP (Model View View Model). Джон Гусман оголосив про цей шаблон у 2005 році. Це найбільш рекомендований шаблон для розробки програм Android. Цей шаблон походить від шаблону MVC і вирішує проблеми попередніх шаблонів MV (X). Розглянемо його принцип роботи:

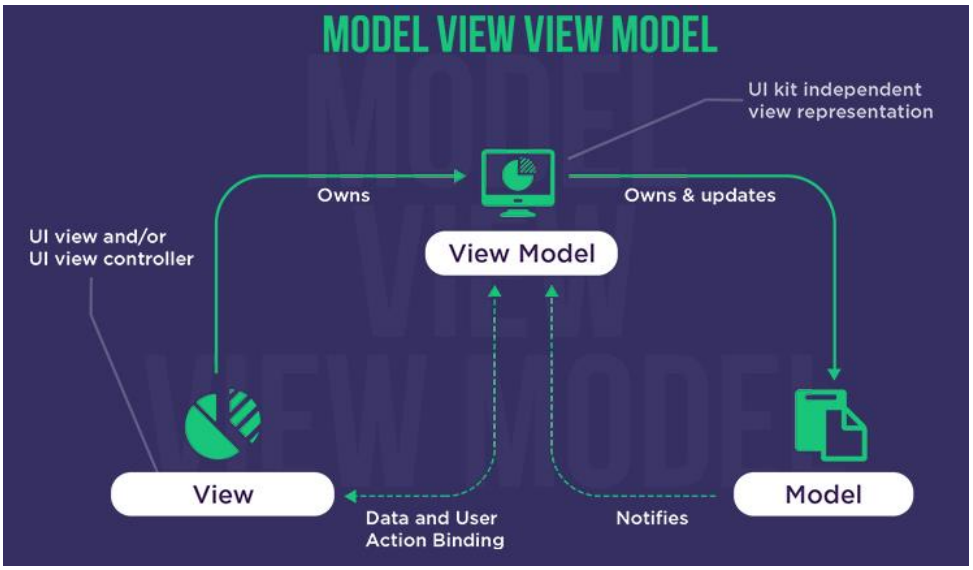


Рисунок 1.13 – Принцип роботи Model-View-ViewModel

Шаблон MVVM включає три компоненти: Model, View і ViewModel. Тут View Model виступає посередником. View Model ініціює зміни в моделі, а також оновлює сама себе на основі оновленої моделі. Крім того, він включає прив'язування даних і дій користувача, як шаблон MVP Supervising Controller, але між View & View Model замість View & Model. Це прив'язує View до відповідного оновлення на основі посилання на View Model:

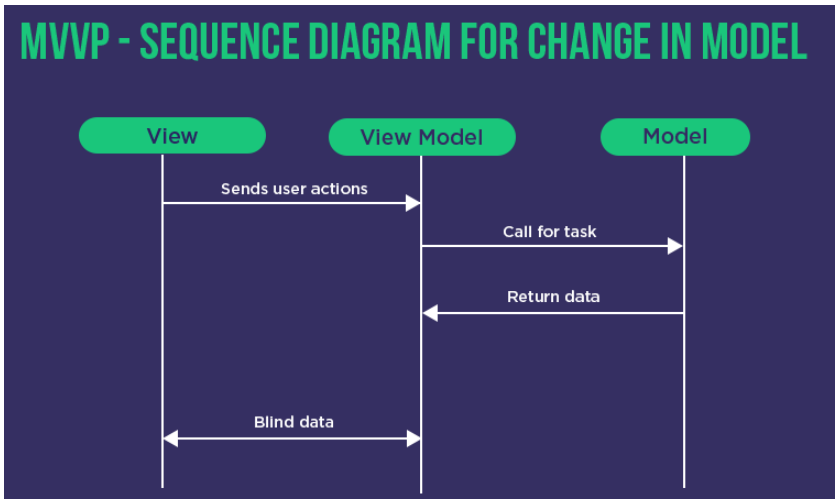


Рисунок 1.14 – Діаграма послідовності змін в MVVM

Переваги перед шаблоном дизайну MVP

- Оскільки односторонній зв'язок від View до ViewModel зменшує кількість рядків коду, необхідні для синхронізації View і ViewModel.

Недолік шаблону проектування MVVM

- Окрім переваг зв'язування даних, у деяких випадках це є недоліком. Цей механізм вимагає значних ресурсів пам'яті, а також стає слабким місцем через експлойт витоку пам'яті.
- На відміну від MVP, тут View відповідає за зміну стану

1.10 Тестування мобільних додатків

Тестування мобільних додатків – це процес, за допомогою якого програмне забезпечення, розроблене для портативних мобільних пристроїв,

перевіряється на його функціональність, зручність використання та узгодженість. Тестування мобільних додатків може бути автоматизованим або ручним типом тестування. Мобільні програми або поставляються попередньо встановленими, або можуть бути встановлені з платформ розповсюдження мобільного програмного забезпечення. У 2015 році глобальний дохід від мобільних додатків склав 69,7 мільярдів доларів США, а до 2020 року, за прогнозами, складе 188,9 мільярда доларів США.

Bluetooth, GPS, датчики та Wi-Fi є одними з основних технологій, які використовуються в носових пристроях. Відповідно, тестування мобільних додатків зосереджується на польовому тестуванні, зосередженні на користувачах і на тих областях, де апаратне та програмне забезпечення необхідно тестувати в унісон [8].

1.10.1 Типи тестування мобільних додатків

Ринок мобільних додатків є конкурентним, що означає попит на кращу зручність використання, безпеку та якість мобільних додатків. Тестування мобільних додатків допомагає виявити недоліки рішення для мобільних додатків і уточнити його для цільової аудиторії. Тому важливо подумати про різні типи мобільного тестування, щоб з’ясувати різні точки зору для оцінки потенційної ефективності програми. Розглянемо деякі з цих типів:

- *Тестування юзабіліті (Usability Testing)*

Зі списку типів тестування веб-додатків тестування юзабіліті ідеально підходить для визначення того, як додаток полегшує користувачам досягнення своїх цілей. Цей тест передбачає призначення певним учасникам конкретних реалістичних сценаріїв використання програми. Тестування юзабіліті також є надійним на основі збору прямих відгуків від кінцевого користувача.

Як результат, у процесі тестування немає упередженості, а також забезпечується покращення виділених областей. Тестування юзабіліті також допомагає перевірити, чи є дизайн інтуїтивно зрозумілим, і підкреслює простоту використання та обслуговування клієнтів.

- *Тестування продуктивності (Performance Testing)*

Тестування продуктивності також є важливим варіантом тестування мобільних додатків, який перевіряє швидкість, стабільність і реагування програми в умовах навантаження. Основна мета тесту продуктивності – переконатися, що програма ідеально відповідає цілям продуктивності [8]. Крім того, він також усуває вузькі місця продуктивності перед запуском програми. Вузькі місця – це процеси в загальних функціях систем, які сповільнюють або гальмують загальну продуктивність. Поширені типи тестів продуктивності включають навантажувальне тестування, об'ємне тестування, тестування на замочування, тестування на стрибки та стрес-тестування.

- *Тестування безпеки (Security Testing)*

Безпека є однією з головних проблем майже кожного власника мобільного додатка в наш час. Як повідомляється, 80 відсотків користувачів частіше видаляють програму через проблеми з безпекою. Тому дуже важливо зосередитися на тестуванні безпеки мобільних додатків.

Деякі програми, такі як програми для подорожей, вимагають особисту інформацію користувачів для різних транзакцій. Якщо програма вимагає чогось подібного, важливо, була надана гарантія конфіденційності, цілісності та автентичності програми. Тому компанії з тестування безпеки або команда тестування QA також повинні зосередитися на безпеці даних і поведінці додатків у випадку різних схем дозволів пристрою.

Рисунок 1.15 – Види тестування мобільних додатків

- *Тестування на переривання (Interruption Testing)*

Тестування на переривання корисно для перевірки поведінки програми в перерваному стані перед відновленням попереднього стану [7]. Загальні приклади перерв можуть включати в себе вхідні телефонні дзвінки або SMS, будильники, push-повідомлення від мобільних додатків, низький або повний заряд акумулятора, втрату та відновлення з'єднання з мережею, а також підключення або вимкнення під час заряджання. У разі переривання програма зазвичай працює у фоновому режимі і має повернутися до попереднього стану до переривання.

- *Ручне тестування (Manual Testing)*

Ручне тестування є одним із перевірених підходів для ретельного визначення складності тестування мобільних додатків. Ручне тестування допомагає переконатися, що кінцевий продукт працює ідеально відповідно до очікуваних очікувань. Він особливо застосовний у випадках використання, які можуть бути не надто очевидними. Фахівці з тестування QA можуть працювати через короткі проміжки часу для перевірки програми, яка може дати надійні результати. За даними компаній, що проводять ручне тестування, поширені типи ручних тестів включають дослідницьке тестування, тести фізичного інтерфейсу та складні тести.

- *Тестування на сумісність (Compatibility Testing)*

Одним з найважливіших варіантів тестування мобільних додатків є тестування на сумісність. Це тип нефункціонального тестування, який важливий для забезпечення роботи мобільного додатка в різних операційних системах, програмах, пристроях, конкретних внутрішніх специфікаціях обладнання та мережевих середовищах. Тестування на сумісність перевіряє сумісність мобільного додатка з різними операційними системами та їх відповідними версіями.

Також перевіряє сумісність мобільного додатка з різними пристроями, браузерами, мережами та пов'язаними параметрами. Тестування на сумісність поділяється на дві категорії: тестування зворотної сумісності та пряме

тестування сумісності. Тестування на зворотну сумісність включає тестування поведінки мобільного додатка зі старішими версіями програмного забезпечення. З іншого боку, сумісність із пересиланнями передбачає тестування поведінки мобільних додатків з новими, а також бета-версіями програмного забезпечення.

- *Тестування локалізації (Localization Testing)*

Тестування локалізації є важливою вимогою для мобільних додатків, які націлені на певну географічну причину. Важливо перевірити мобільний додаток на відповідність специфічним мовним та культурним аспектам відповідного регіону. Деякі з основних областей, які перевіряє тестування на локалізацію, включають місцеву валюту, використання відповідних форматів дати та часу часового поясу, різні вимоги місцевих нормативних актів, а також текст і інтерфейс користувача.

- *Функціональне тестування (Functional Testing)*

Тестування функціональних мобільних додатків допомагає перевірити, чи функціональні можливості програми відповідають необхідним цілям. Такий тип тестування зосереджений значною мірою на головній цілі та потоці мобільного додатка. Під час роботи над сервісами функціонального тестування перевіряється, що функції мобільного додатка відповідають необхідним специфікаціям і мають високу швидкість реагування.

Тестування функціональності перевіряє, чи можна програму правильно запустити та встановити. Крім того, він також перевіряє легкість реєстрації та входу, а також відтворення push-повідомлень, а також належне функціонування текстових полів і кнопок.

- *Тестування установки (Installation Testing)*
- Тестування встановлення або впровадження ідеально підходить для перевірки того, що мобільний додаток правильно встановлено та видалено. Крім того, тестування встановлення є життєво важливим для забезпечення безперебійності оновлень і відсутності помилок. Тестування встановлення

також перевіряє результати, коли користувачі не оновлюють певний мобільний додаток.

- *Автоматичне тестування (Automated Testing)*

Як було вказано раніше, ручне тестування — це комплексний варіант тестування мобільних додатків. Проте деякі тести якості мобільних додатків є надзвичайно складними та виснажливими. У таких випадках Служби автоматизації тестування мобільних додатків представлені в образі та ідеально налаштованим та ефективно виконаним автоматизованим тестуванням разом із ручними тестами, які можуть допомогти у забезпеченні якості разом із швидшим випуском кращих продуктів. Багато повторень тестування мобільних додатків, безумовно є виснажливою вправою. Однак, якщо забезпечити функціональність свого мобільного додатка на всіх фронтах, це підвищує шанси на задоволення клієнтів.

1.11 Постановка задачі

На основі проведеного літературного аналізу за тематикою роботи необхідно виконати наступне:

- обрати інструментарій для розробки мобільного додатку;
- розробити архітектуру і дизайн;
- розробити мобільний додаток для платформи iOS, який допоможе користувачеві залишатись продуктивним протягом дня, та концентруватись на поставлених задачах;
- протестувати працездатність додатку, зробити відповідні висновки.

2. МЕТОДИКА ВИРІШЕННЯ ПОСТАВЛЕНИХ ЗАДАЧ

Як було зазначено в першому розділі, розробка мобільних додатків це достатньо складний процес не тільки з точки зору бізнесу, але з точки зору самого процесу написання коду, розробки інтерфейсу тощо.

Коли епоха створення програмного забезпечення тільки набирала обертів, розробники не мали оптимальних інструментів які б допомагали їм виконувати свою роботу, не витрачаючи дорогоцінний час на пошук, або налагодження спеціальних програм, або взагалі на створення таких з нуля.

Сьогодні ситуація виглядає набагато ліпше, і провідні компанії, наприклад такі як Apple і Google випускають власні інструменти для створення мобільних додатків для своїх платформ iOS та Android.

Оскільки в рамках даної роботи за мету взято розробку мобільного додатку для операційної системи iOS, будуть розглянуті лише інструменти які актуальні для даної платформи.

2.1 Мова програмування Swift

Swift є результатом останніх досліджень мов програмування в поєднанні з десятиліттям досвіду створення платформ Apple. Названі параметри виражені в чистому синтаксисі, що робить API в Swift ще легшим для читання та обслуговування. Також в Swift не обов’язково вводити крапку з комою наприкінці рядка. Виведені типи роблять код чистішим і менш схильним до помилок, а модулі усувають заголовки та надають простір імен. Щоб найкраще підтримувати міжнародні мови та емодзі, рядки коректні за Unicode і використовують кодування на основі UTF-8 для оптимізації продуктивності для різноманітних випадків використання [1]. Управління пам’яттю здійснюється автоматично за допомогою жорсткого, детермінованого підрахунку посилань, що зводить використання пам’яті до мінімуму без накладних витрат на збирання сміття [12]. Ви навіть можете писати

одночасний код за допомогою простих, вбудованих ключових слів, які визначають асинхронну поведінку, роблячи ваш код більш читабельним і менш схильним до помилок.

```
struct Player {
    var name: String
    var highScore: Int = 0
    var history: [Int] = []

    init(_ name: String) {
        self.name = name
    }
}

var player = Player("Tomas")
```

Рисунок 2.1 – Приклад написання об’єкту Player мовою Swift

Swift усуває цілі класи небезпечного коду. Змінні завжди ініціалізуються перед використанням, масиви та цілі числа перевіряються на переповнення, пам’ять автоматично керується, а ексклюзивний доступ до пам’яті захищає від багатьох помилок програмування. Синтаксис налаштований так, щоб було легко визначити ваш намір – наприклад, прості трисимвольні ключові слова визначають змінну (var) або константу (let) [12]. Swift значною мірою використовує типи значень, особливо для часто використовуваних типів, таких як масиви та словники. Це означає, що коли ми робимо копію чогось що має такий тип, то впевнені, що в іншому місці це не буде змінено.

Ще одна функція безпеки полягає в тому, що за замовчуванням об'єкти Swift ніколи не можуть бути нульовими. Насправді, компілятор Swift завадить вам створити або використовувати нульовий об’єкт з помилкою під час компіляції. Це робить написання коду набагато чистішим і безпечнішим, а також запобігає величезній категорії збоїв під час виконання ваших програм. Однак є випадки, коли нуль є дійсним і доречним. Для цих ситуацій Swift має

інноваційну функцію, відому як опціональний тип (Optional). Необов'язковий параметр може містити нуль, але синтаксис Swift змушує безпечно працювати з ним за допомогою символу «?», щоб вказати компілятору, що розумієте поведінку та безпечно обробляєте її. Swift має багато інших функцій, які роблять код більш виразним [12]:

- *Генерики (Generics), які є потужними та простими у використанні*

Загальне програмування – це спосіб написання функцій і типів даних, роблячи мінімальні припущення щодо типу даних, що використовуються. Узагальнені засоби Swift створюють код, який не враховує основні типи даних, що дозволяє використовувати елегантні абстракції, які створюють чистіший код з меншою кількістю помилок. Це дозволяє написати функцію один раз і використовувати її для різних типів.

Наприклад, стандартна незагальна функція під назвою swapTwoInts(_:_:), яка міняє місцями два значення Int (Рис 2.2):

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

Рисунок 2.2 – Функція swapTwoInts міняє місцями два цілих числа

Функція swapTwoInts(_:_:) корисна, але її можна використовувати лише зі значеннями Int. Якщо необхідно поміняти місцями два значення String або два значення Double, потрібно написати більше функцій, таких як функції swapTwoStrings(_:_:) і swapTwoDoubles(_:_:), як вказано на рисунку 2.3:

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

Рисунок 2.3 – Функції що міняють місцями типи даних String та Double

Як можна помітити, тіла функцій swapTwoInts(_:_:), swapTwoStrings(_:_:) і swapTwoDoubles(_:_:) ідентичні. Єдина відмінність — це тип значень, які вони приймають (Int, String і Double).

- *Внутрішня обробка помилок*

Обробка помилок – це процес реагування на помилки та відновлення після них у програмі. Swift забезпечує першокласну підтримку для викиду, лову, поширення та маніпулювання помилками, які можна виправити, під час виконання. Деякі операції не завжди завершують виконання або виробляють корисний результат. Необов'язкові параметри використовуються для представлення відсутності значення, але коли операція зазнає збою, часто корисно зрозуміти, що спричинило збій, щоб код міг відповідним чином реагувати.

- *Структури та класи*

Структури та класи є гнучкими конструкціями загального призначення, які стають будівельними блоками коду програми. Розробник визначає властивості та методи для додавання функціональних можливостей до структур і класів, використовуючи той самий синтаксис, що використовується для визначення констант, змінних і функцій.

На відміну від інших мов програмування, Swift не вимагає створення окремих файлів інтерфейсу та реалізації для користувацьких структур і класів. У Swift ви, як розробник, визначаєте структуру або клас в одному файлі, а зовнішній інтерфейс цього класу або структури автоматично стає доступним для використання іншим кодом.

- *Розширення протоколів*

Swift дозволяє визначати поведінку самих протоколів, а не глобальних функцій або окремих відповідностей.

- *Керування пам'яттю*

Swift використовує автоматичний підрахунок посилань (ARC) для відстеження та керування використанням пам'яті вашою програмою. У більшості випадків це означає, що керування пам'яттю «просто працює» у

Swift, і вам не потрібно думати про керування пам’яттю самостійно. ARC автоматично звільняє пам'ять, яку використовують екземпляри класу, коли ці екземпляри більше не потрібні.

- *Перерахування (Enums)*

Перерахування визначає загальний тип для групи пов’язаних значень і дозволяє працювати з цими значеннями безпечним для типу способом у коді. Перерахування в Swift набагато гнучкіші, і їм не потрібно надавати значення для кожного випадку перерахування. Якщо значення (відоме як вихідне значення) надається для кожного випадку перерахування, значення може бути рядком, символом або значенням будь-якого цілого чи типу з плаваючою комою [1].

Крім того, випадки перерахування можуть вказувати пов’язані значення будь-якого типу, які зберігатимуться разом із кожним різним значенням регістру, як це роблять об’єднання в інших мовах. Можна визначити загальний набір пов’язаних випадків як частину одного перерахування, кожен з яких має інший набір значень відповідних типів, пов’язаних з ним.

Перерахування в Swift самі по собі є першокласними типами. Вони використовують багато функцій, які традиційно підтримуються лише класами, наприклад обчислювані властивості, щоб надати додаткову інформацію про поточне значення перерахування, і методи екземплярів для забезпечення функціональності, пов’язаної зі значеннями, які представляє перерахування. Перерахування також можуть визначати ініціалізатори для надання початкового значення регістру; можуть бути розширені, щоб розширити їх функціональність за межі їх початкової реалізації; і може відповідати протоколам для забезпечення стандартної функціональності.

- *Менеджер пакетів*

Менеджер пакетів Swift – це кросплатформний інструмент, який можна використовувати для створення, запуску, тестування та упаковки бібліотек і виконуваних файлів Swift.

- *Замикання*

Замикання – це автономні функціональні блоки, які можна передавати та використовувати у коді. Замикання в Swift подібні до блоків у C і Objective-C, а також до лямбда функцій в інших мовах програмування.

Замикання можуть захоплювати та зберігати посилання на будь-які константи та змінні з контексту, в якому вони визначені. Це відомо як закриття цих констант і змінних. Swift керує всіма процесами управління пам’яті автоматично.

З самого початку, Swift створювався, щоб бути швидким. Використовуючи неймовірно високопродуктивну технологію компілятора LLVM, код Swift перетворюється на оптимізований машинний код, який отримує максимум користі від сучасного обладнання. Синтаксис і стандартна бібліотека також були налаштовані, щоб зробити найбільш очевидний спосіб написання коду також найкращим, незалежно від того, працює він у годиннику на зап’ясті або на кластері серверів.

Swift є наступником мов C і Objective-C. Він включає примітиви низького рівня, такі як типи, керування потоками та оператори. Він також надає об’єктно-орієнтовані функції, такі як класи, протоколи та генерики, надаючи розробникам Cocoa та Cocoa Touch необхідну продуктивність та потужність [1].

Хоча Swift використовують багато нових програм на платформах Apple, він також використовується для нового класу сучасних серверних програм. Swift ідеально підходить для використання в серверних програмах, які потребують безпеки під час виконання, продуктивності компіляції та невеликого обсягу пам’яті. Щоб керувати напрямком розробки та розгортання серверних додатків Swift, спільнота сформувала робочу групу Swift Server. Першим продуктом цих зусиль був SwiftNIO, кросплатформена асинхронна мережева програма, керована подіями, для серверів і клієнтів високопродуктивних протоколів. Він служить основою для створення додаткових серверно-орієнтованих інструментів і технологій, включаючи журналювання, метрики та драйвери баз даних, які активно розробляються.

2.2 Xcode

Xcode – це програма, яку розробники використовують для створення програм, під будь які пристрої компанії Apple, таких як iPhone, iPad, Apple TV та WatchOS. Він використовує мову програмування Swift або Objective-C для розробки додатків [1].

Оскільки, Xcode – інтегроване середовище розробки, то воно також містить багато інших додаткових інструментів, які потрібні для розробки додатків. Це один з найпопулярніших інструментів для створення мобільних додатків, які можна використовувати на різних пристроях і операційних системах компанії Apple, а також розробляти серверну частину за допомогою мови програмування Swift [7].

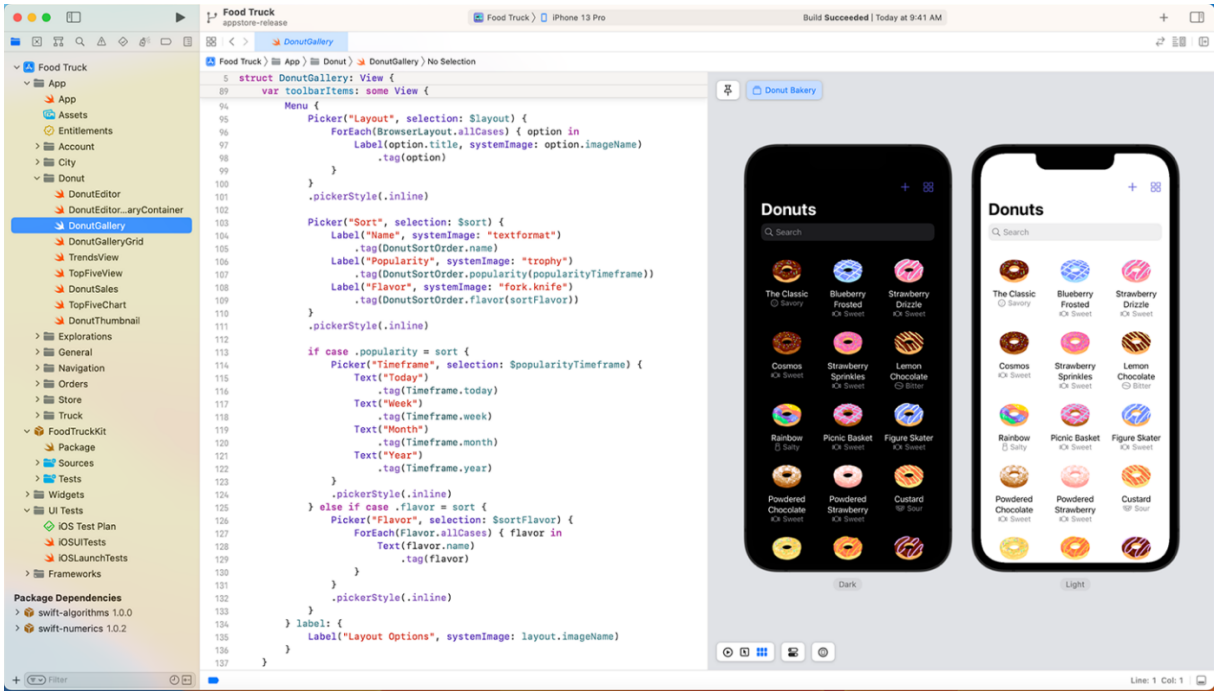


Рисунок 2.4 – Зовнішній вигляд Xcode

Використовуючи Xcode, розробники можуть виконувати безліч завдань, починаючи від проектування інтерфейсу користувача, написання коду для додатків, компіляції та тестування коду та перевірки на наявність помилок у коді.

Xcode є невід'ємним елементом, який діє як ядро для всіх інших компонентів, доступних у пакеті Xcode. Він показує файли, над якими триває робота, а також Windows для інших інструментів. Інтерфейс схожий на інші середовища, де код вводиться у файл у головному вікні. IDE також розширює підтримку та гарантує, що коди, введені користувачами, легко зрозуміти, а помилки зведені до мінімуму [7].

Розробники отримують пропозиції щодо того, що вони можуть захотіти ввести в локацію. Він також висвітлює проблеми, коли будь-який із очікуваних символів відсутній або назви функцій введено неправильно. У більшості випадків також пропонуються засоби для вирішення цих проблем.

Розробники також мають можливість залишати відкритими кілька вкладок і перемикатися між цими вкладками. Інтерфейс оновлюється відповідно до файлу, над яким ведеться робота. Також доступне бокове вікно навігації, якие дозволяє переходити від одного файлу до іншого, а також перераховує всі файли та папки, які використовуються для будь-якого конкретного проекту. У процесі розробки коду користувачі мають можливість проводити кілька експериментів з кодом. Є кілька готових до використання проектів, які корисні для навчання користувачів [8].

У розробників є безліч варіантів мов програмування, коли справа доходить до написання коду в Xcode. Список мов програмування, які підтримує Xcode, варіюється від Swift, AppleScript, C, C++, Objective C, Python тощо. Серед усіх цих мов Apple настійно рекомендує мову Swift для розробки додатків для усіх своїх платформ.

2.3 SwiftUI та UIKit

Основним фреймворком для розробки інтерфейсів користувача в Xcode до 2019 року був UIKit. Цей фреймворк розроблений компанією Apple який до появи мови програмування Swift був написаний на Objective-C та з перших версій Swift поступово мігрував на цю мову програмування.

UIKit пропонує багато вбудованих класів для побудови інтерфейсу користувача, одним основним з яких є UIView, об’єкт, який керує вмістом прямокутної області на екрані. Від цього класу унаслідуються майже усі інші класи які відповідають за елементи інтерфейсу користувача, такі як UIButton, UILabel, UITableView та інші. Усі об’єкти які мають в своїх іменах префікс UI є елементами фреймворку UIKit.

UIKit забезпечує необхідну інфраструктуру для iOS додатків. Він надає архітектуру вікон і представлення для реалізації інтерфейсу, інфраструктуру обробки подій для доставки Multi-Touch та інших типів введення у додатку, а також основний цикл виконання, необхідний для керування взаємодією між користувачем, системою та додатком. Інші функції, які пропонує фреймворк, включають підтримку анімації, підтримку документів, підтримку малювання та друку, інформацію про поточний пристрій, керування текстом та відображенням, підтримку пошуку, підтримку спеціальних можливостей, підтримку розширень програми та керування ресурсам [7].

У 2019 році на щорічній конференції розробників WWDC, компанія Apple представила новий фреймворк для розробки інтерфейсів користувача, який називається SwiftUI. Відтоді фреймворк розвивається швидкими темпами. На відміну від UIKit, SwiftUI є кросплатформним фреймворком. Завдяки SwiftUI Apple пропонує розробникам рішення для швидкого створення додатків.

Концепції, що лежать в основі SwiftUI, дуже відрізняються від UIKit. Фреймворк пропонує все необхідне для створення користувацьких інтерфейсів, таких як списки, стеки, кнопки, засоби вибору та багато інших компонентів, які знайомі розробникам в UIKit. Він також надає інструменти, необхідні для створення користувацьких переглядів, додавання анімації та інтеграції жестів [8].

Незважаючи на те, що SwiftUI сильно відрізняється від UIKit, варто взяти до уваги, що SwiftUI покладається на цей фреймворк для створення інтерфейсу користувача. Під капотом у деяких випадках, додаток продовжує використовувати компоненти UIKit, включаючи UIView. Ключова відмінність

від UIKit полягає в тому, що SwiftUI визначає інтерфейс користувача декларативно, а не імперативно.

Використовуючи UIKit, ми створюємо представлення для побудови ієрархії представлень інтерфейсу користувача. SwiftUI працює не так. SwiftUI надає розробникам API, щоб оголошувати або описувати, як повинен виглядати інтерфейс користувача. SwiftUI перевіряє декларацію або опис користувальницького інтерфейсу та перетворює його в інтерфейс користувача додатку. SwiftUI робить усю важку роботу автоматично.

SwiftUI – це більше, ніж фреймворк. Він інтегрований в Xcode, що дозволяє розробникам створювати інтерфейси користувача набагато швидше, ніж будь-коли раніше. Ви можете редагувати користувальницький інтерфейс своєї програми в коді або у візуальному редакторі, який автоматично відображає те, у що перекладається ваш код [1].

Xcode надає можливість попереднього перегляду інтерфейсу користувача, який створює розробник. Це можливо шляхом компіляції та виконання написаного коду та відображення його в попередньому перегляді. Попередній перегляд можна зробити інтерактивним, усуваючи необхідність запускати програму в симуляторі або на пристрої. Це значно прискорює розробку інтерфейсу користувача [1].

2.4 Vapor

Vapor – це веб-фреймворк який дозволяє писати серверні програми, API веб-програм і HTTP-сервери на Swift. Vapor написаний на Swift, який є сучасною, потужною та безпечною мовою, що має ряд переваг у порівнянні з більш традиційними мовами серверів. Vapor складається з ряду пакетів, включаючи Leaf – механізм шаблонів для розробки інтерфейсу – і Fluent, Swift Object Relational Mapping (ORM) фреймворк з рідними асинхронними драйверами баз даних [3]. У рамках даної роботи, серверна частина проекту написана саме за допомогою Vapor.

2.5 Fluent

Fluent – це платформа ORM для Swift. Він використовує переваги сильних типів Swift, щоб забезпечити простий у використанні інтерфейс для бази даних. Використання Fluent зосереджується на створенні типів моделей, які представляють структури даних у базі даних. Ці моделі потім використовуються для виконання операцій створення, читання, оновлення та видалення замість написання необроблених запитів [4].

2.6 Swifty Beaver

Платформа SwiftyBeaver є першою в світі платформою для запису логів для Swift. Це забезпечує дуже зручне автоматичне, високозахищене отримання та аналіз логів, аналітичних даних користувачів і пристроїв, які програма створює під час випуску. Платформа складається з таких компонентів:

- SwiftyBeaver Framework для створення логів у вашій програмі під час розробки та випуску;
- SwiftyBeaver Crypto Cloud для зберігання та синхронізації зашифрованих логів;
- Додаток SwiftyBeaver для Mac для завантаження, пошуку й аналізу логів, надісланих із вашої програми.

2.7 Sketch

Sketch – це інструмент векторної графіки, який використовується для дизайну інтерфейсу користувача для настільних і мобільних пристроїв, макетів і прототипів, і сьогодні він вважається галузевим стандартом. Також за допомогою Sketch можна редагувати зображення. Sketch має наступні переваги.

- Простий і легкий у навчанні, інтуїтивно зрозумілий інтерфейс;

- Дозволяє створювати проекти для кількох пристроїв;
- Клієнти можуть попередньо переглянути, що станеться, коли вони взаємодіють з дизайном настільного комп’ютера, ноутбука, телефону або планшета;
- Корисні функції спільної роботи, особливо за допомогою синхронізації та спільного доступу Sketch Cloud;
- Існує набір плагінів, які можна використовувати з Sketch (Abstract, Craft, Flinto тощо);
- Доступна ціна.

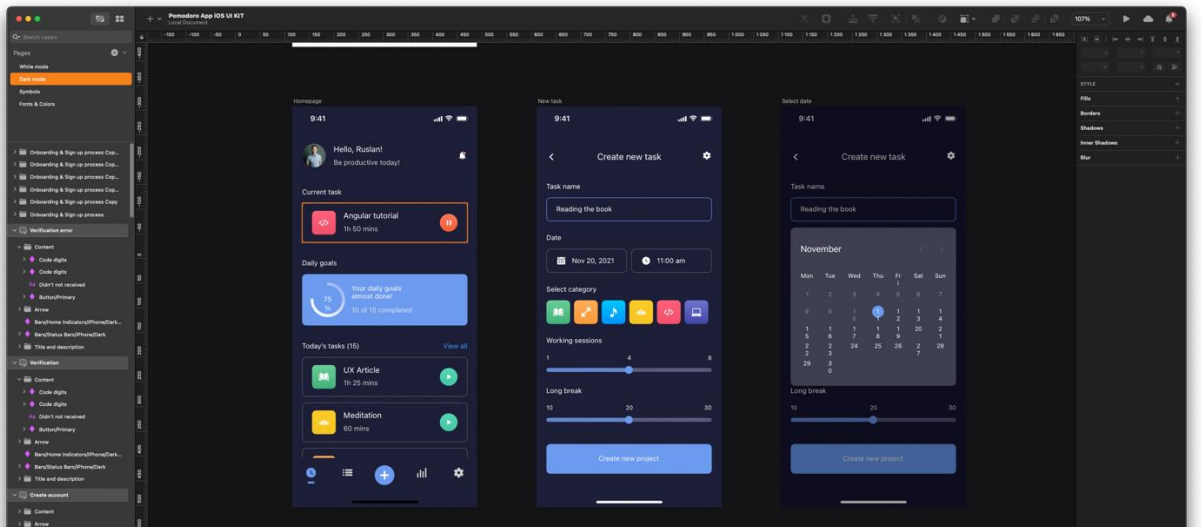


Рисунок 2.7 – Головне вікно Sketch

Sketch має багато переваг у порівнянні з Photoshop. Наприклад, він працює так само, як Photoshop, але за нижчою ціною і займає менше місця на диску. Деякі вважають, що Sketch може навіть обігнати Photoshop в якийсь момент, принаймні, коли мова йде про провідний інструмент створення прототипів. Велика причина, чому Sketch може бути більш вправним у дизайні екрана, ніж Photoshop, полягає в тому, що саме для цього він був спеціально розроблений – це молодший, сучасніший інструмент, який був спеціально створений.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ МОБІЛЬНОГО ДОДАТКУ

Як було зазначено, метою цієї роботи є створення мобільного додатку, якій допоможе користувачам бути більш продуктивними протягом дня, менше відволікатись, а також боротись з прокрастинацією.

Прокрастинація – це дія невиправданого й добровільного відкладення чи відкладення чогось, незважаючи на знання того, що це матиме негативні наслідки. Слово походить від латинського слова procrastinatus, яке само походить від префікса pro-, що означає «вперед», і crastinus, що означає «завтра». Це звичайний людський досвід, пов’язаний із затримкою виконання повсякденних справ або навіть відкладенням важливих завдань, таких як відвідування зустрічі, подання звіту про роботу чи академічного завдання чи розгляд стресової проблеми з партнером.

Тож оскільки наше повсякденне життя невід’ємне від сучасних технологій, вони у рамках даної роботи, а саме у вигляді мобільного додатку, прослужать для вирішення даної проблеми.

З огляду на обмеження ресурсів, у процесі розробки даного мобільного додатку не були використані усі принципи, які були описані в першому розділі, але була обрана оптимальна архітектура проекту, яка в майбутньому дозволить легко додавати нові функції у мобільний додаток, а також дозволяє без великого рефакторингу коду проводити його тестування.

Особливістю мобільного додатку є те, що окрім озвученої мети, він також буде виконувати роль органайзера який дозволить користувачеві впорядкувати свій денний графік, та завжди пам’ятати про заплановані справи.

3.1 Структура серверної частини додатку

Проект мобільного додатку складається с трьох частин – це дизайн інтерфейсу користувача, серверна частина, та сам мобільний додаток для платформи iOS.

Уся інформація про користувача буде зберігатись на сервері, до якої він зможе отримати доступ за допомогою клієнта (мобільного додатку) ввівши свій імейл і пароль.

Рішення зберігання інформації на сервері має як свою плюси так і недоліки. Серед переваг можна відмітити те, що користувач може отримати доступ до своїх задач з будь якого пристрою, а також перейшовши на веб-сайт, якщо такий буде розроблений в майбутньому, а значить фактично отримати доступ з будь якого пристрою, а не тільки з iOS. Серед недоліків слід зазначити те, що користувач повинен постійно мати доступ до мережі інтернет, та ця проблема може бути частково вирішена за допомогою хешування.

Серверна частина проекту, побудована на базі фреймоворку Vapor з використанням архітектурного шаблона MVC, оскільки він ідеально підходить для розробки API через яке мобільний додаток буде комунікувати з сервером. API побудовані на базі архітектурного стилю REST.

Передача стану репрезентації (REST) — це архітектурний стиль програмного забезпечення, який був створений для управління проектуванням і розробкою архітектури для всесвітньої павутини. REST визначає набір обмежень для того, як повинна вести себе архітектура розподіленої гіпермедійної системи масштабу Інтернет, наприклад Web. Архітектурний стиль REST підкреслює масштабованість взаємодії між компонентами, уніфіковані інтерфейси, незалежне розгортання компонентів і створення багатошарової архітектури для полегшення кешування компонентів, щоб зменшити затримку, забезпечити безпеку та інкапсулювати застарілі системи.

REST використовується в індустрії програмного забезпечення і є широко прийнятим набором рекомендацій щодо створення надійних веб-API. Веб-API, що задоволняє обмеженням REST, неофіційно описується як RESTful. Веб-API RESTful зазвичай базуються на методах HTTP для доступу до ресурсів через параметри, закодовані URL-адресою, і використання JSON або XML для передачі даних.

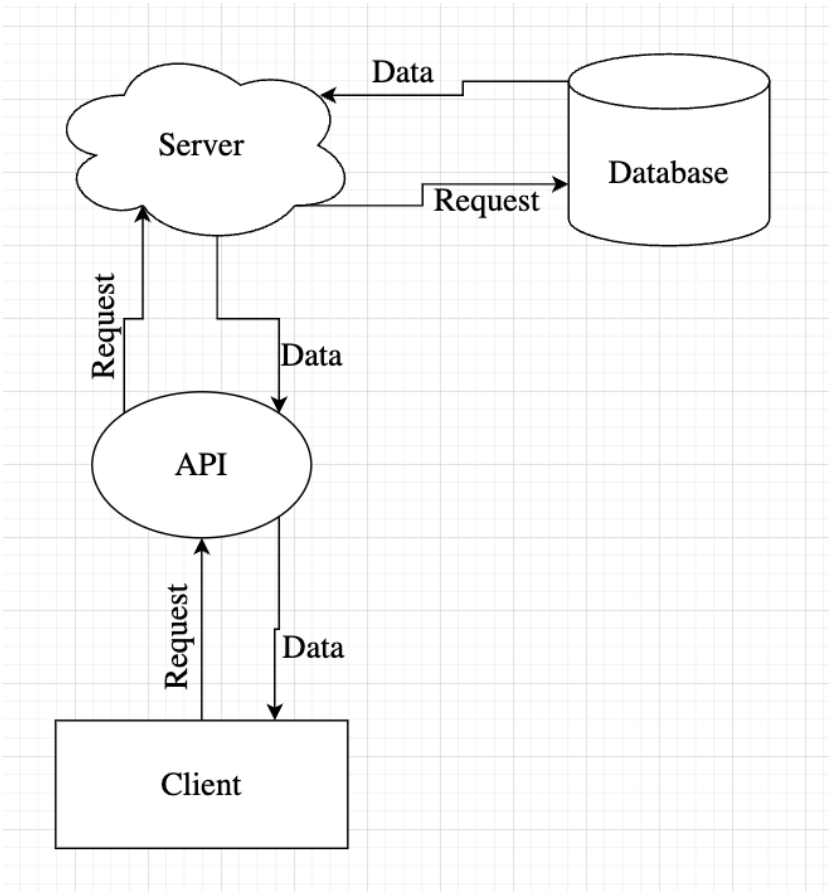


Рисунок 3.1 – Принцип роботи API

3.1.1 Робота з базою даних

У якості бази даних, з якою працює сервер, було обрано PostgresSQL. PostgreSQL – це широко розповсюджена система керування базами даних з відкритим серцевим кодом. Вона базується на мові SQL і підтримує багато можливостей стандарту SQL:2011

Сервер був розгорнутий у Docker-контейнері на локальній машині. Це звільняє нас від ручного налаштування бази даних оскільки Docker усе робить автоматично, та при необхідності є можливість задати додаткові параметри за допомогою Docker CLI.

Використовування фреймворку Fluent дозволяє нам не використовувати мову запитів SQL для створення таблиць у базі даних, а робити це безпосередньо у коді за допомогою Swift.

Для зберігання даних у БД, за допомогою Fluent спочатку потрібну створити модель яка. Моделі представляють дані, що зберігаються в таблицях

або колекціях у вашій базі даних. Моделі мають одне або кілька полів, які зберігають кодовані значення. Усі моделі мають унікальний ідентифікатор. Обгортки властивостей використовуються для позначення ідентифікаторів, полів і відносин.

```swift
//
//  Created by Руслан Кукса on 09.04.2022.
//

import Fluent
import Vapor

final class User: Model, Content {
    static var schema = "user"

    @ID(key: .id)
    var id: UUID?

    @Field(key: "name")
    var name: String

    @Field(key: "email")
    var email: String

    @Field(key: "password")
    var password: String

    @Children(for: \.$user)
    var tasks: [Task]

    init() {}

    init(
        id: UUID? = nil,
        name: String,
        email: String,
        password: String
    ) {
        self.id = id
        self.name = name
        self.email = email
        self.password = password
    }
}
```

Рисунок 3.2 – Модель користувача

Як видно на рисунку 3.2 клас User задовольняє протоколу Model який є частиною Fluent, та вказує на те, що наша модель є репрезентацією об’єкту в базі даних.

Усі моделі вимагають статичну властивість scheme, яка не може змінюватись. Цей рядок посилається на назву таблиці або колекції, яку представляє ця модель. Під час запиту цієї моделі дані будуть витягуватися та зберігатися в схемі під назвою "user". Усі моделі повинні мати властивість id, визначену за допомогою оболонки властивості @ID. Це поле однозначно

ідентифікує екземпляри моделі. За замовчуванням властивість @ID має використовувати спеціальний ключ .id, який перетворює відповідний ключ для базового драйвера бази даних. Для SQL це "id", а для NoSQL це "_id". @ID також повинен мати тип UUID. Це єдине значення ідентифікатора, яке зараз підтримується всіма драйверами баз даних. Fluent автоматично генеруватиме нові ідентифікатори UUID під час створення моделей. @ID має необов'язкове (Optional) значення, оскільки незбережені моделі можуть ще не мати ідентифікатора.

Після того як була створена модель, слід подбати про міграції.

Міграції – це як система контролю версій бази даних. Кожна міграція визначає зміну в базі даних і спосіб її скасування. Змінюючи свою базу даних шляхом міграції, ми створюємо послідовний, перевірений і доступний спосіб розвитку ваших баз даних з часом.

```
struct CreateUser: AsyncMigration {
    func prepare(on database: Database) async throws {
        try await database.schema("user")
            .id()
            .field("name", .string, .required)
            .field("email", .string, .required)
            .field("password", .string, .required)
            .unique(on: "email")
            .create()

    }

    func revert(on database: Database) async throws {
        try await database.schema("user").delete()
    }
}
```

Рисунок 3.3 – Міграція об’єкту User

Метод *prepare* вносить зміни до наданої бази даних. Це можуть бути зміни в схемі бази даних, наприклад додавання чи видалення таблиці чи

колекції, поля чи обмеження. Вони також можуть змінювати вміст бази даних, наприклад, створювати нові екземпляри моделі, оновлювати значення полів або виконувати очищення.

За допомогою методу *revert* можна скасувати ці зміни, якщо це можливо. Можливість скасувати міграцію може полегшити створення прототипів і тестування. Вони також надають резервний план, якщо розгортання у виробництво не пройде за планом.

Все це актуально не тільки для об’єкту User а і для усіх інших об’єктів які зберігаються у БД. Наприклад об’єкту Task, якій представляю собою структуру задачі, яку буде створювати користувач за допомогою мобільного додатку.

3.1.2 Взаємодія з API

Як було сказано на початку, API побудовано за архітектурним шаблоном MVC. Це означає що для кожного об’єкта бази даних з яким оперує користувач за допомогою запитів, був створений окремий контроллер. Оскільки за RESTful API запити можуть відрізнятись за типом (GET, POST, DELETE, PUT, PATCH), було створено окремий метод для кожного типу запиту.

Так наприклад якщо користувач хоче створити нове завдання, він надсилає запит с типом POST, якій буде оброблений методом *create* у об’єкті TaskController (Рис 3.4)

```
    func create(_ req: Request) async throws -> TaskDTO {
        //1
        let token = try req.auth.require(Payload.self)

        //2
        let model = try req.content.decode(CreateTaskRequest.self, using: decoder)
        let task = Task(
            title: model.title,
            date: model.date,
            totalSessions: model.totalSessions,
            longBreakDuration: model.longBreakDuration,
            category: model.category,
            userID: token.userId
        )

        //3
        try await task.save(on: req.db)
        return task.convertToDTO()
    }
```

Рисунок 3.4 – Обробка запиту на створення нового завдання

Цей процес можна описати наступним чином:

1) Перевіряємо чи був разом із запитом надісланий JWT токен. JSON Web Token – це стандарт токена доступний на основі JSON, стандартизованого в RFC 7519. Як правило, використовується для передачі даних для аутентифікації в клієнт-серверних програмах. Токені створені сервером, підписані секретним ключем і передаються клієнту, який надає цей токен для підтвердження особи. Якщо ідентифікація пройшла успішно, то значить, що користувач авторизований і запит може бути оброблений у іншому випадку, буде повернута помилка.

2) Декодуємо надіслану інформацію у запиті в об’єкт CreateTaskRequest, у разі якщо це неможливо зробити буде повернута помилка. Після чого створюємо новий об’єкт Task на базі model. Варто помітити, що userId береться не з поля model а з поля token, оскільки саме token зберігає інформацію про користувача у закодованому вигляді.

3) Зберігаємо об’єкт у базі даних.

3.2 Дизайн мобільного додатку

Дизайн мобільного додатку було розроблено у програмі Sketch з урахуванням усіх особливостей платформи iOS, а також рекомендаціями та практиками які описані у Human Interface Guidelines. Дизайн був розроблений з урахуванням як світлої так і темної теми (рис 3.5)

Найкращі практики при розробці додатків для iOS включають у себе:

- Допомагайте людям зосередитися на основних завданнях і вмісті, обмежуючи кількість елементів керування на екрані, роблячи доступними другорядні деталі та дії з мінімальною взаємодією.
- Легко адаптуйте до змін зовнішнього вигляду – наприклад, орієнтації пристрою, темного режиму та динамічного типу – дозволяючи людям вибирати конфігурації, які найкраще підходять для них.
- Брати до уваги взаємодії, як люди зазвичай тримають свій пристрій. Наприклад, людям, як правило, легше та зручніше дістатися до елемента керування, коли він розташований у середній або нижній частині дисплея, тому особливо важливо, щоб люди проводили пальцем, щоб перейти назад або почати дії в рядку списку.
- З дозволу користувача інтегруйте інформацію, доступну через можливості платформи, таким чином, щоб покращити роботу, не вимагаючи від людей вводити дані. Наприклад, можна приймати платежі, забезпечувати безпеку за допомогою біометричної автентифікації або пропонувати функції, які використовують місцезнаходження пристрою.

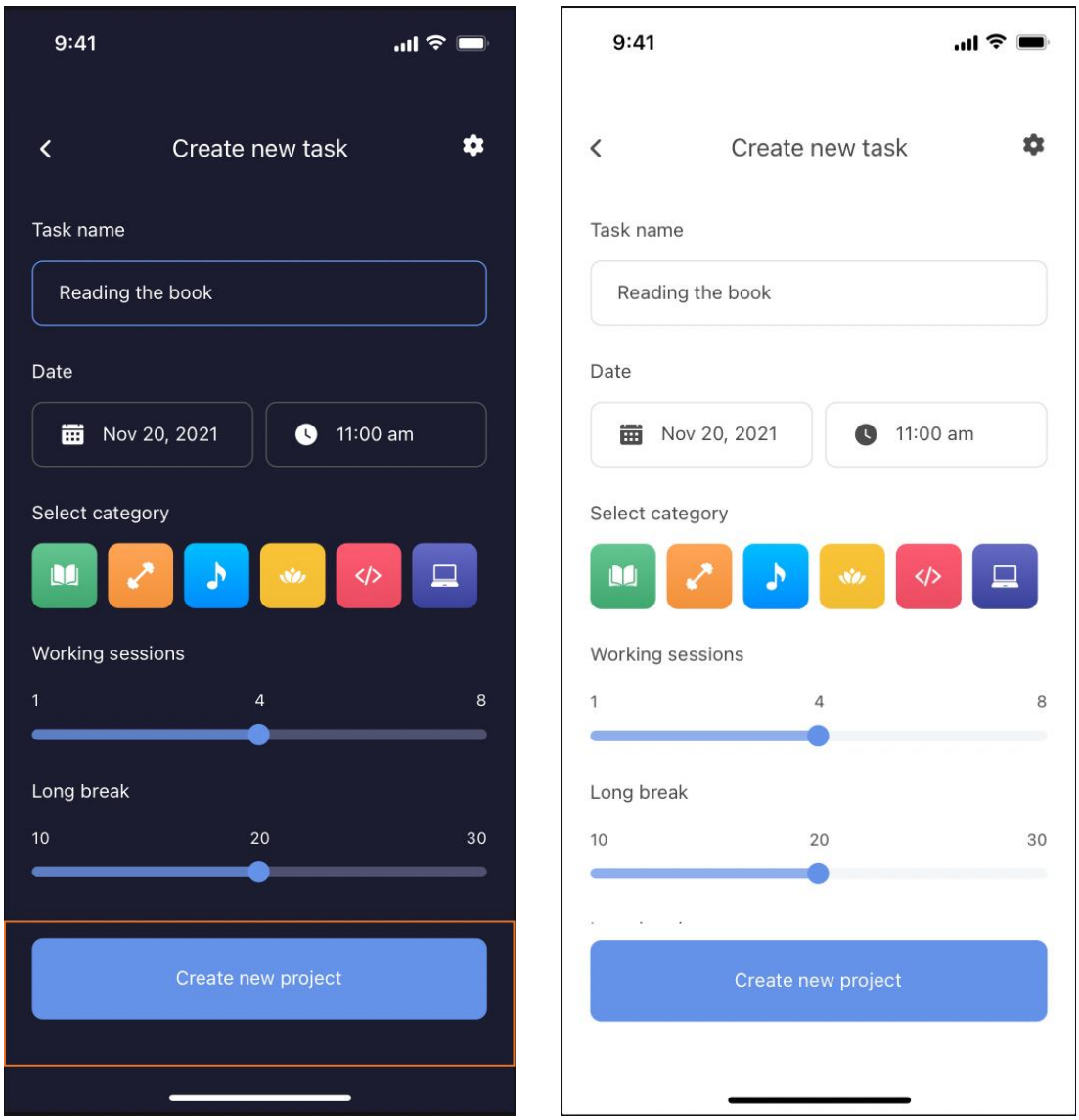


Рисунок 3.5 – Екран створення нової задачі

3.3 Опис мобільного додатку

Для того щоб підвищити продуктивність користувачів, мобільний додаток реалізує техніку помодоро.

Техніка помодоро – це система управління часом, яка заохочує людей працювати з часом, який у них є, а не проти нього. Використовуючи цей метод, ви розбиваєте свій робочий день на 25-хвилинні частини, розділені п’ятихвилинними перервами. Ці інтервали називаються помодорами. Приблизно після чотирьох помідорів ви робите більшу перерву приблизно на 15-20 хвилин.

Ідея цієї техніки полягає в тому, що таймер вселяє відчуття невідкладності. Замість того, щоб відчувати, що у вас є нескінченний час у робочому дні, щоб зробити щось, а потім зрештою витрачати ці дорогоцінні робочі години на відволікання, ви знаєте, що у вас є лише 25 хвилин, щоб досягти максимального прогресу у виконанні завдання.

Якщо відкинути екрани реєстрації або авторизації, то додаток складається з п’яти основних екранів:

- *Екран поточних задач на сьогодні*

Цей екран користувач бачить одразу після реєстрації або авторизації. На ньому відображається список поточних задач на сьогодні у вигляді невеликих комірок. Запустити або призупинити 25-хвилинний таймер задачі можна натиснувши кнопку у комірці. Слід зауважити, що у користувача не може бути більше однієї активної задачі. При натисканні на саму комірку, а не на кнопку, здійснюється перехід на екран детяльног огляду задачі де користувач бачить скільки часу у нього залишилось до кінця поточної сесії, також там відображаються елементи керування таймером. Сесію можна зупинити, перезапустити або поставити на паузу.

- *Екран з розкладом*

На цьому екрані відображаються дні у потомному місяці, натиснувши на будь яке число, користувач побачить список запланованих задач на цей день.

- *Екран статистики*

На цьому екрані відображається статистика того наскільки користувач був продуктивним. Користувач може отримати статистику за день, тиждень, або місяць

- *Екран налаштувань*

На екрані налаштувань, користувач може увімкнути або вимкнути push-повідомлення які інформують користувача про заплановані задачі, або коли час на таймері поточної задачі спливає. Також він може змінити звуковий сигнал який звучить при закінчені задачі або перерви. Змінити інформацію про

себе, включити або відключити темну тему, вийти зі свого облікового записи тощо.

- *Екран створення задач*

На цьому екрані користувач може створювати нові задачі. Вказувати їх назву, дату та час на коли він хоче запланувати задачу, вказати кількість 25-хвилинних інтервалів для неї, а також часовий інтервал для довгої перерви між сесіями.

Інтерфейс користувача додатку побудований за допомогою новітнього фреймворку SwiftUI. Навігація між основними екранами здійснюється за допомогою TabBarView – це елемент інтерфейсу (рис 3.6) який зазвичай розташовується у нижній частині екрану, і включає у себе ту кількість кнопок яка відповідає кількості екранів між якими потрібно перемикатися.

Активний екран на TabBarView зазвичай підсвічується іншим кольором на відміну від неактивних.

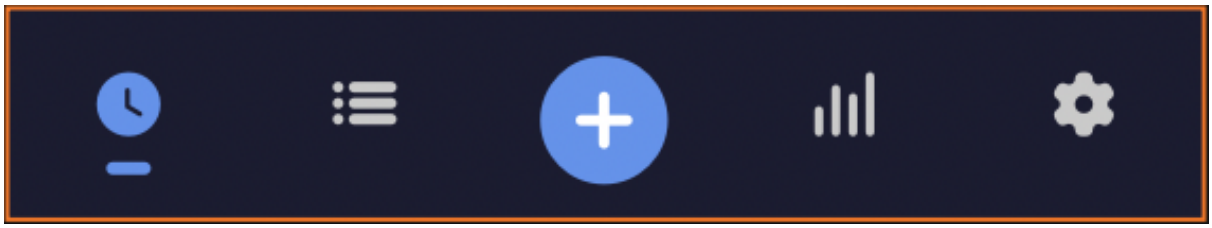


Рисунок 3.6 – TabBarView

Сам додаток побудований на базі архітектурного шаблону MVVM оскільки він дозволяє чітко відділити усю бізнес логіку від View, що робить код чистішим, також більш адаптованим до написання юніт тестів.

Також сервіси, що відповідають за комунікацію с сервеною частиною були винесені у окремі об’єкти, які за допомогою техніки Dependency Injection додаються у ту чи іншу ViewModel.

У процесі роботи були дотримані основні принципи SOLID, що дозволяє у майбутньому легко додавати у проект нові функції.

ВИСНОВКИ

У рамках даної роботи були розглянуті основні етапи і принципи побудови мобільних додатків. Було проведено знайомство з процесами на кожному із етапів розробки, починаючи від ідеї до вибору архітектурного підходу у написанні коду. Зрозуміло, що створення мобільних додатків – це складний і ресурсоємкий процес, який потребує не менших зусиль ніж розробка повноцінних настільних програм.

Також у рамках цієї роботи було створено власний мобільних додаток з урахуванням усієї отриманої інформації. Цей додаток спрямований на підвищення продуктивності користувачів, та боротьби с прокрастинацією.

Мобільний додаток було розроблено для платформи iOS з урахуванням усіх сучасних технологій у цій області. Таких як фреймворк SwiftUI та мови програмування Swift.

Також були розглянуті аспекти розробки серверної частини та принципи побудови і взаємодії API через яке додаток взаємодіє із сервером.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Apple Developer Documentation [Електронний ресурс] // Apple Inc. – 1976. – Режим доступу до ресурсу: https://developer.apple.com/

2. Swift Documentation [Електронний ресурс] - Режим доступу до ресурсу: https://swift.org/

3. Vapor Swift Web Framework [Електронний ресурс] – Режим доступу до ресурсу http://vapor-docs-site.s3-website.eu-west-2.amazonaws.com/

4. Fluent ORM Framework for Swift [Електронний ресурс] – Режим доступу до ресурсу https://docs.vapor.codes/fluent/overview/

5. Hacking with Swift [Електронний ресурс] – Режим доступу до ресурсу https://www.hackingwithswift.com/

6. Swift by Sundell [Електронний ресурс] – Режим доступу до ресурсу: https://www.swiftbysundell.com/

7. Ray Wenderlich. [Електронний ресурс] - Режим доступу до ресурсу: https://www.raywenderlich.com

8. Medium [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/tag/swift

9. Siliconithub [Електронний ресурс] – Режим доступу до ресурсу: https://siliconithub.com/native-vs-hybrid-app/

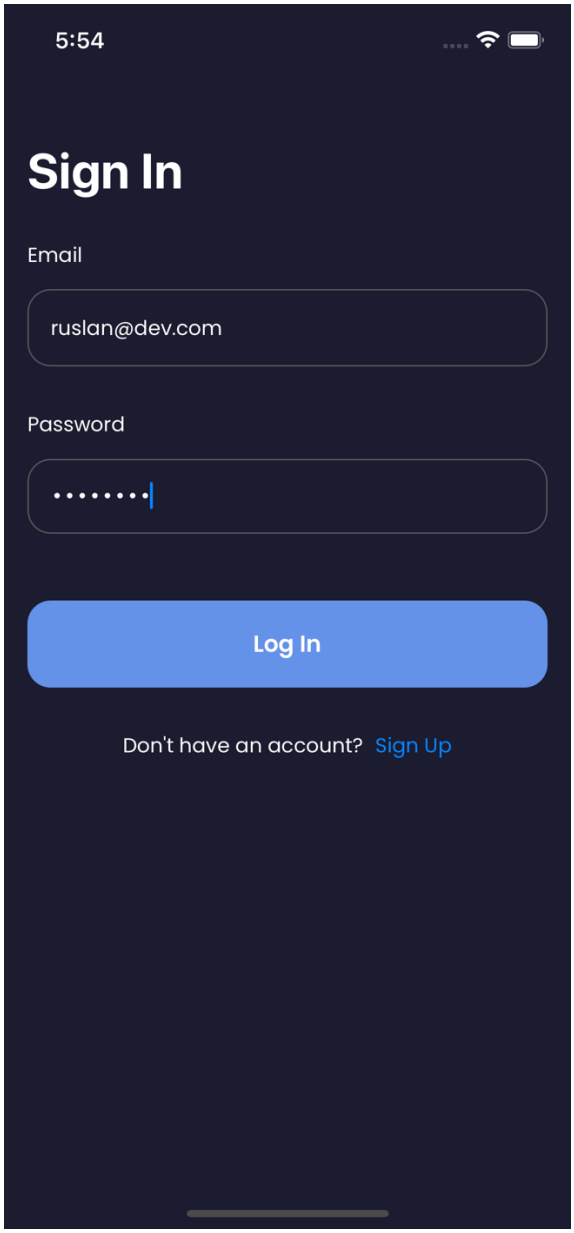
10. NiX-United [Електронний ресурс] – Режим доступу до ресурсу: https://nix-united.com/blog/

11. Clariontech [Електронний ресурс] – Режим доступу до ресурсу: https://www.clariontech.com/blog/evaluating-design-patterns-for-mobile-development

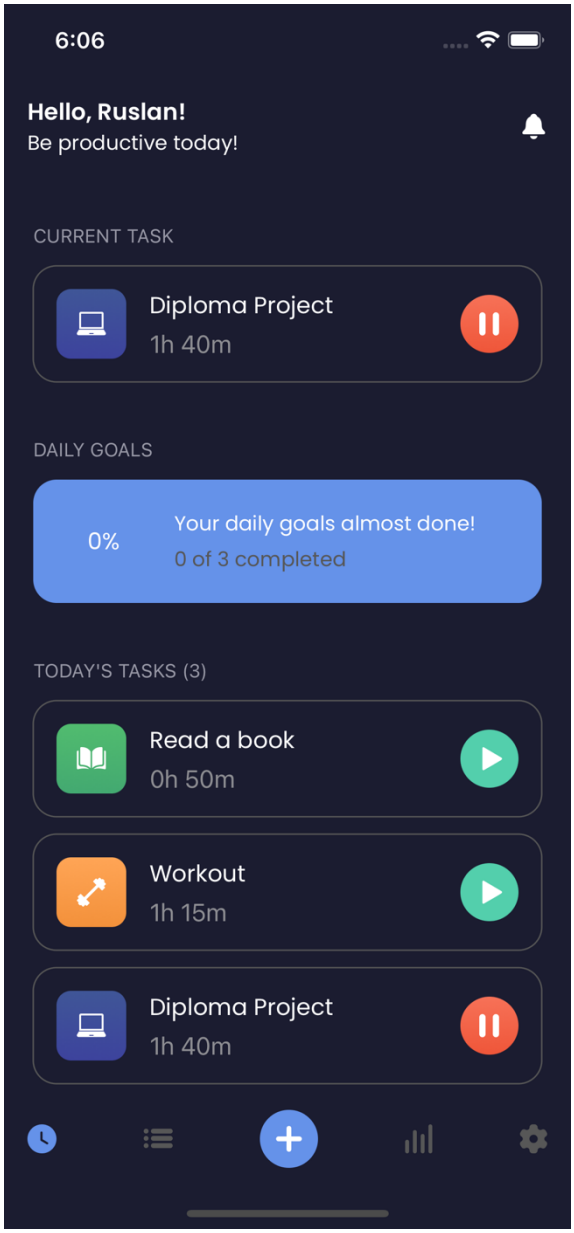
12. Swift Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.swift.org/documentation/

ДОДАТОК

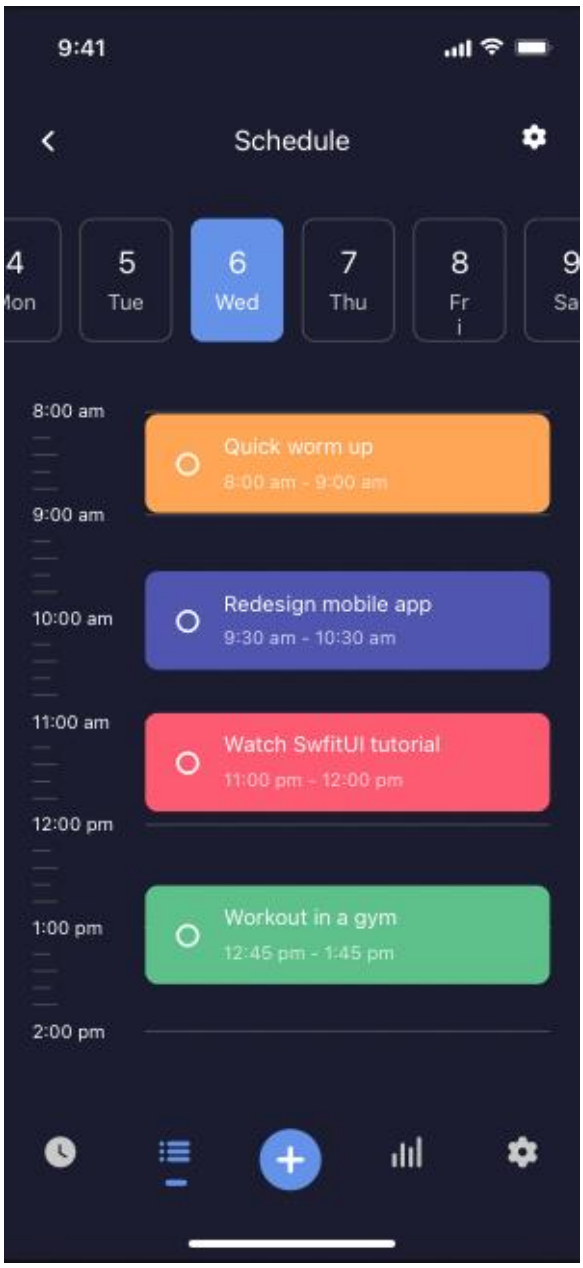
У додатку представлені скріншоти з розробленого мобільного додатку.



Екран авторизації

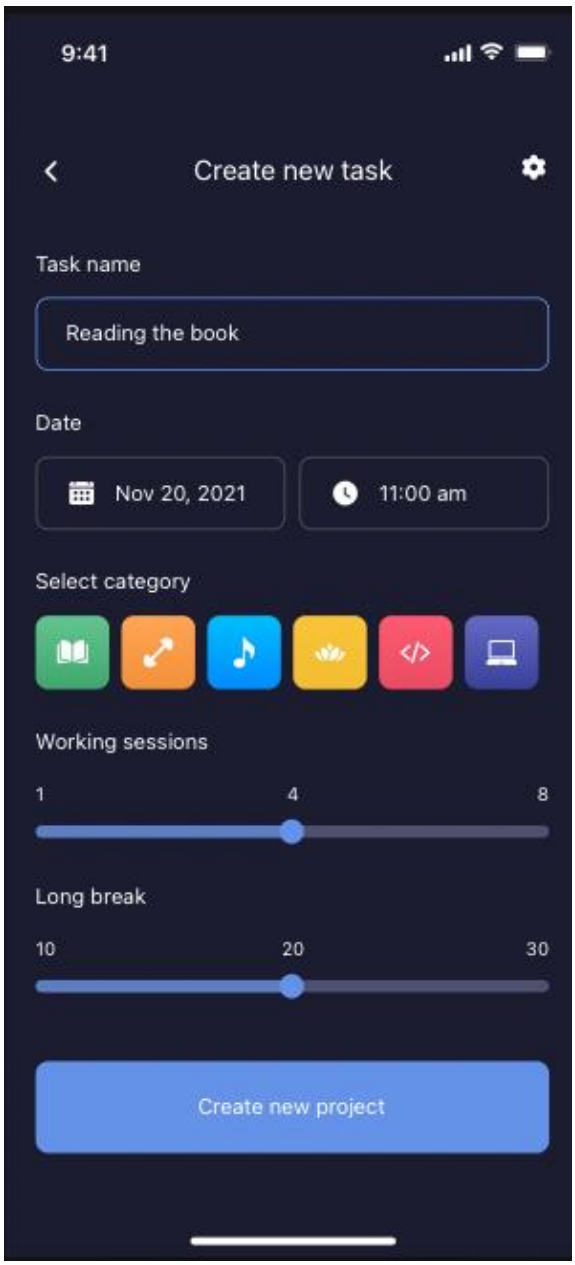


Список задач на сьогодні

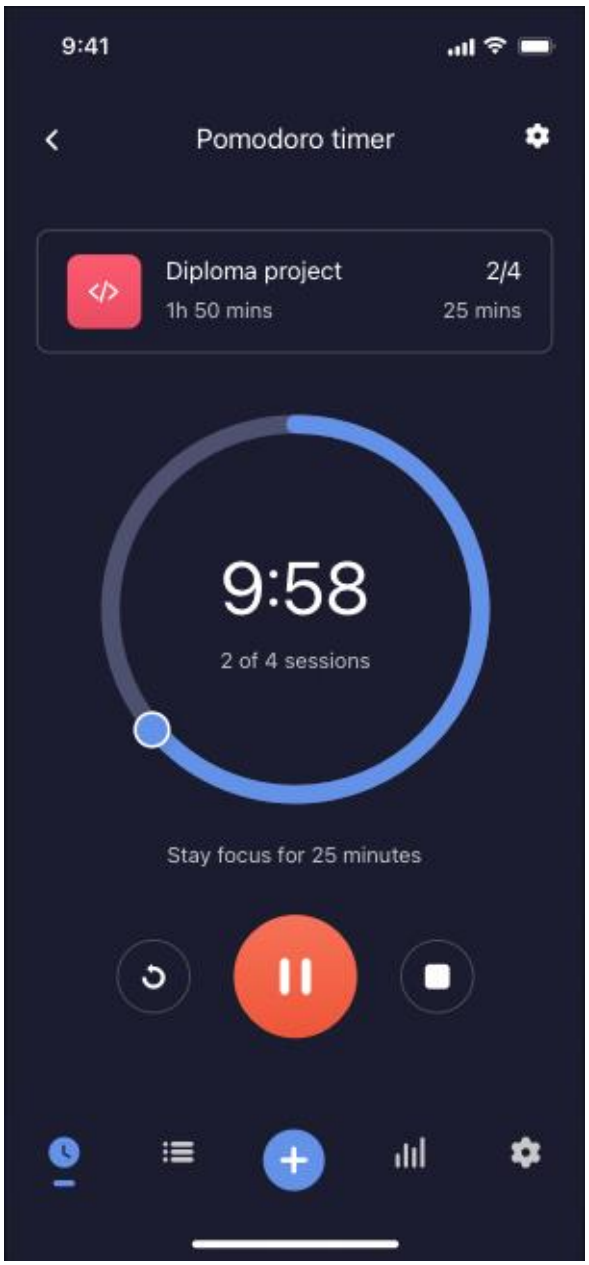


Розклад на обрану дату

Статистика



Створення нової задачі



Детальний огляд задачі

Серцевий код:

https://bitbucket.org/ruslankuksa/workspace/projects/TAS