МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ   
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Программное обеспечение информационной безопасности мобильных систем

Специальность 1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА**

по дисциплине «Программирование мобильных систем»

Тема «Мобильное приложение магазина страйкбольного снаряжения»

**Исполнитель**

студент(ка) 3 курса 8 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П. А. Заруцкий

подпись, дата

**Руководитель**

ассистент \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н. И. Уласевич

должность, ученая степень, ученое звание подпись, дата

Допущен(а) к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н. И. Уласевич

подпись дата инициалы и фамилия

Реферат

Пояснительная записка курсового проекта содержит 34 страницы пояснительной записки, 8 иллюстраций, 7 источников литературы, 2 приложения.

Цель курсового проекта – разработка программных модулей, которые будут корректно взаимодействовать между собой и смогут обеспечить безошибочное функционирование приложения для салона оптик.

В первой главе проводится анализ предметной области и формирование функционала.

Вторая глава посвящена процессу проектирования системы и описанию технологий, которые будут использованы в проекте.

В третьей главе описаны подробности реализации.

В четвертой главе описано тестирование и проверка работоспособности получившихся приложений.

В пятой главе описывается руководство пользователя.

В заключении приведены результаты проделанной работы.

Содержание

[Реферат 2](#_Toc166849754)

[Введение 5](#_Toc166849755)

[1 Анализ и моделирование предметной области, формирование функциональных требований к проектируемому мобильному приложению 6](#_Toc166849756)

[1.1 Исследование предметной области, выявление основных проблем 6](#_Toc166849757)

[1.2 Разработка функциональных требований 6](#_Toc166849758)

[1.3 Технические требования 7](#_Toc166849759)

[1.4 Итоги раздела 8](#_Toc166849760)

[2 Проектирование приложения 9](#_Toc166849761)

[2.1 Архитектура 9](#_Toc166849762)

[2.2 Сущности в базе данных и связи между ними 9](#_Toc166849763)

[2.3 Серверная часть приложения 10](#_Toc166849764)

[2.3.1 Уровни. API, Service, Repository 11](#_Toc166849765)

[2.3.2 Объекты, использующиеся на различных уровнях, и их особенности 11](#_Toc166849766)

[2.3.3 Безопасность. Аутентификация и авторизация 12](#_Toc166849767)

[2.3.4 Хранение фото 12](#_Toc166849768)

[2.4 Клиентская часть приложения 13](#_Toc166849769)

[2.4.1 Виджеты, управление состоянием виджетов, декларативный UI, фрагменты и диалоги, навигация 13](#_Toc166849770)

[2.4.2 Внедрение зависимостей 14](#_Toc166849771)

[2.4.3 Логика использования приложения 15](#_Toc166849772)

[2.4.4 Дизайн и расположение элементов 15](#_Toc166849773)

[2.5 Веб-приложение 16](#_Toc166849774)

[2.6 Итоги раздела 16](#_Toc166849775)

[3 Реализация приложения 17](#_Toc166849776)

[3.1 Серверная часть 17](#_Toc166849777)

[3.1.1 Конфигурация 17](#_Toc166849778)

[3.1.2 Реализация REST API. Контроллеры 17](#_Toc166849779)

[3.1.3 Сервисы. 18](#_Toc166849780)

[3.1.4 Репозитории. Взаимодействие с Firestore 18](#_Toc166849781)

[3.1.5 DTO. Конвертация разных типов объектов 19](#_Toc166849782)

[3.1.6 Аутентификация и авторизация 19](#_Toc166849783)

[3.2 Клиентская часть 21](#_Toc166849784)

[3.2.1 Виджеты, разметка, диалоги. 21](#_Toc166849785)

[3.2.2 Внедрение зависимостей 21](#_Toc166849786)

[3.3 Итоги раздела 22](#_Toc166849787)

[4 Тестирование, проверка работоспособности и анализ полученных результатов 23](#_Toc166849788)

[5 Руководство по использованию 24](#_Toc166849789)

[Заключение 28](#_Toc166849790)

[Список литературы 29](#_Toc166849791)

[Приложение А 30](#_Toc166849792)

[Приложение Б 35](#_Toc166849793)

Введение

Цель данного курсового проекта – разработка мобильного приложения магазина страйкбольного снаряжения, сервера для него, а также веб-клиента.

Задачи курсового проекта:

1. Изучение программных средств, аналогичных теме курсового проекта

Данная задача позволит изучить аналоги данному программному средству, разобраться в их минусах, плюсах. Найти отличия между аналогами и данным программным средством.

1. Проектирование приложения

Проектирование приложения даёт возможность увидеть предварительную концепцию приложения, позволяет спланировать разработку программного средства и соответствующе распределить ресурсы и время, которые будут потрачены на разработку приложения.

1. Реализация программного средства

В курсовом проекте будет реализовано приложение для магазина страйкбольного снаряжения. Данная тема актуальна в век информационных технологий, где люди часто делают выбор в пользу мобильных приложений для онлайн покупок, так как у каждого есть компьютер или смартфон с доступом в Интернет.

1. Тестирование разработанного продукта

Важный этап разработки, позволяющий проверить функциональность приложения и его реакцию на исключительные ситуации, например, ввод данных неподходящего формата, неверные данные авторизации и др.

1. Разработка руководства пользователя

Данная задача подразумевает разработку руководства пользователя, призванное помочь пользователю разобраться с работой приложения и его основным функционалом. Это является важной частью разработки, поскольку определённые пользователи заинтересованы перед использованием ознакомиться с руководством пользователя, чтобы изучить функционал приложения.

После прохождения всех перечисленных этапов, должно получиться полностью рабочее и отлаженное приложение.

1 Анализ и моделирование предметной области, формирование функциональных требований к проектируемому мобильному приложению

1.1 Исследование предметной области, выявление основных проблем

В современном мире мобильные приложения для совершения онлайн покупок занимают значительную часть рынка. Людям гораздо проще ознакомиться с каталогом товаров и узнать информацию об их наличии, не выходя из дома, тем более у каждого под рукой есть компьютер или смартфон, имеющие доступ в сеть Интернет. За счет онлайн магазинов увеличивается клиентская база, состоящая преимущественно из молодежи. Причем популярностью пользуются не только приложения, в которых собраны различные категории товаров. Многие приложения нацелены на определённую сферу, одной из которых является сфера военно-тактических игр.

Изучив предлагаемые сервисом Google Play мобильные приложения на данную тематику, было замечено, что на рынке нет приложений, предоставляемых магазинами страйкбольного снаряжения.

Как отмечалось ранее, главным плюсом данного мобильного приложения является то, что воспользоваться им сможет любой желающий, ведь для этого понадобиться только смартфон с доступом в Интернет, имеющийся практически у любого человека. Таким образом, взяв во внимание текущее положение дел, было принято решение разработать подобное приложение, главной целью которого будет возможность просмотра каталога, а также покупка желаемого снаряжения.

1.2 Разработка функциональных требований

После того как выяснены основные проблемы в области и определён фундаментальный функционал, необходимо его конкретизировать и преобразовать в функциональные требования, а также разработать дополнительные функции для того, чтобы улучшить пользовательский опыт.

В ходе использования приложения, пользователей можно разделить на две роли: сами клиенты, имеющие доступ к просмотру каталога и администраторы, которые могут добавлять, изменять и удалять каталог товаров.

Помимо самого приложения для смартфона, необходимо сделать сервер, который будет ответственен за реализацию всей логики, поддержку аутентификации и авторизации и взаимодействию с базой данных.

В итоге был сформирован следующий функционал:

* просмотр каталога товаров;
* добавление продуктов в каталог;
* добавление товаров в избранное;
* возможность изменения общей информации о товаре (название, брэнд, модель, описание, фото и т.д.);
* возможность удаления товаров;
* поиск и фильтр товаров;
* наличие нескольких ролей: «пользователь», «администратор».

1.3 Технические требования

Практически все функции мобильного приложения требуют взаимодействия с сервером, который в свою очередь взаимодействует с базой данных, следовательно, для работы приложения требуется стабильное подключение к интернету. От работоспособности перечисленных сервисов целиком зависит работоспособность самого приложения.

Для клиента мобильного приложения, с учётом того, что одной из основных целей является привлечение большего количества пользователей, выбраны системы Android и iOS. Для разработки приложения был выбран фреймворк Flutter от Google. Его главным преимуществом является возможность кроссплатформенно разработки. Нам не нужно разрабатывать нативные приложения под разные операционные системы на разных языках программирования, достаточно знать один Dart. Несмотря на то, что данный фреймворк очень молодой, для него имеется большое количество готовых библиотек, книг, статей и всевозможных инструкций, что, без сомнений, облегчает процесс разработки приложения. Операционная система Android имеет множество версий, чем ниже версия системы, под которую будет разрабатываться приложение, тем больше устройств смогут её запустить, но тем меньше новых функций будет доступно при разработке. Таким образом, оптимальным выбором оказалось версия под номером 26.

Серверная часть приложения будет разработана с использованием Firestore Database, Firebase Storage и Firebase Authentication. Язык программирования, используемый при разработке, - TypeScript. Этот выбор обусловлен простотой использования Firebase, его готовым набором инструментов для работы с данными и аутентификацией пользователей, а также его масштабируемостью.

Firestore Database будет использоваться в качестве основного хранилища данных серверной части приложения. Здесь будут храниться информация о пользователях, их профили и другие сущности, необходимые для функционирования приложения. Firebase Storage будет использоваться для хранения и обработки медиафайлов, таких как изображения или видео, связанных с приложением. Firebase Authentication будет обеспечивать безопасность и аутентификацию пользователей при входе в приложение.

Мобильное приложение будет взаимодействовать с сервером через API, предоставляемый Firebase. Firebase Authentication будет использоваться для обеспечения безопасности и аутентификации пользователей при входе в приложение. Firestore Database будет служить основным хранилищем данных, где будут храниться информация о пользователях, их профили, а также другие сущности, необходимые для функционирования приложения. Firebase Storage будет использоваться для хранения и обработки медиафайлов, таких как изображения или видео, связанных с приложением.

Такой выбор Firebase обусловлен его простотой в использовании, готовым набором инструментов для работы с данными и аутентификацией пользователей, а также его масштабируемостью. Firebase также предоставляет удобное API и SDK для разработки мобильных приложений, что упрощает интеграцию клиентской части с сервером.

Как мобильное приложение, так и сервер не требуют много ресурсов для своей работы, поэтому ставить вопрос о комплектующих, необходимых для работы приложения и сервера, не имеет смысла.

1.4 Итоги раздела

По итогам данного раздела была проанализирована предметная область, в ней выявлены ключевые проблемы, которые необходимо устранить, и на основе этого был сформирован функционал программы, а также определены базовые технологии для его реализации.

2 Проектирование приложения

Проектирование приложения – самая важная часть разработки, ведь именно на этапе проектирования выявляются как мелкие огрехи, так и крупные недоработки, исправление которых на этапе его развёртывания будет крайне затруднительно и накладно. Важность проектирования в данной курсовой работе очень высока ещё и потому, что разрабатываться будут три компонента: мобильное приложение, серверная часть, а также веб-приложение. Необходимо продумать то, как они будут без помех взаимодействовать друг с другом.

2.1 Архитектура

Для разработки подобного типа проекта было решено использовать стандартную трёхуровневую архитектуру, так как такая архитектура позволяет добиться низкой связности уровней, что, в свою очередь, обеспечивает для приложения надёжность, простоту долгосрочной поддержки, а также оперативное устранение неисправностей при необходимости.

Клиент представляет из себя либо мобильное, либо веб-приложение с пользовательским интерфейсом, которое будет совершать REST-запросы на сервер Firebase и обрабатывать их в читаемый для пользователя вид. Также на клиенте реализовывается логика обработки полученных от сервера сообщений об каталога и отображение информации.

Серверная часть будет полностью основываться на Firebase. Firebase предоставляет встроенную поддержку аутентификации и авторизации, поэтому сервер будет обрабатывать эти мероприятия. Все взаимодействие с базой данных также будет осуществляться через Firebase, что упрощает работу с данными.

2.2 Сущности в базе данных и связи между ними

Изучение предметной области и составленный впоследствии функционал показали, что можно выделить 9 коллекций, которые позволяют полностью охватить обозначенный функционал (рис. 2.1). Из них 5 основных:

* пользователь (Users) – коллекция, предназначенная для хранения основных данных о пользователе, таких как имя, телефон, электронная почта, пароль;
* заказы (Orders) – коллекция, которая является подколлекцией коллекции пользователей, предназначенная для хранения заказов;
* продукт (Products) – коллекция, предназначенная для хранения данных о продукте;
* производитель (Brands) – в этой коллекции хранятся данные о производителях;
* категории (Categories) – коллекция, предназначенная для хранения информации о категориях продуктов;

Остальные коллекции вспомогательные, и нужны для того, чтобы поддерживать целостность информации

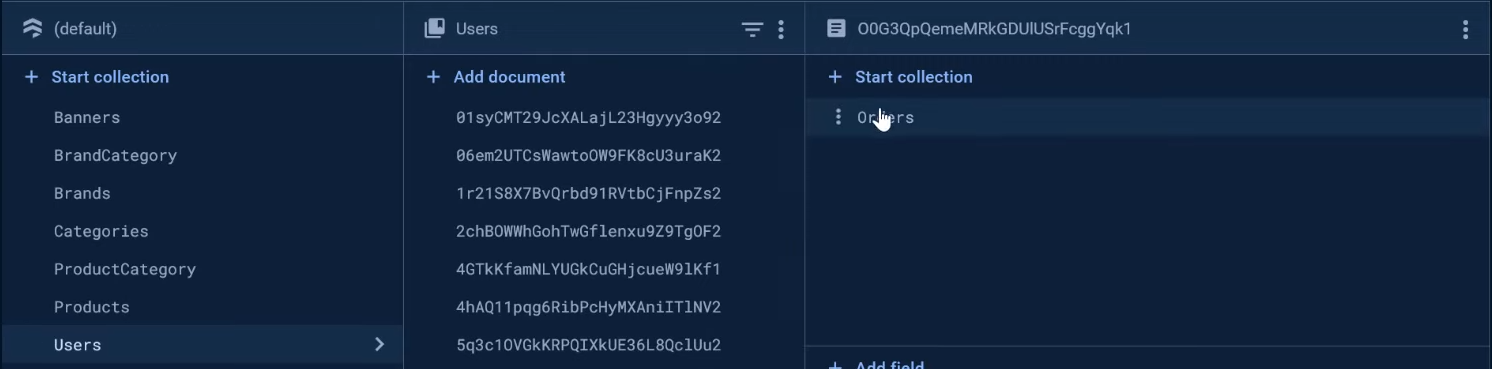


Рисунок 2.1 – Схема коллекций

Отдельно стоит упомянуть способ хранения изображений в базе данных Firestore. Вместо хранения изображений в виде относительных ссылок на файловую систему или абсолютных ссылок на внешние ресурсы, Firestore предлагает использовать Firebase Storage для хранения и обработки изображений.

При загрузке изображений в Firestore, каждое изображение сохраняется в Firebase Storage, которое является облачным хранилищем отдельно от базы данных. В Firestore сохраняется только ссылка на конкретное изображение в Firebase Storage.

В итоге, для хранения изображений в базе данных Firestore используется подход, при котором сами изображения сохраняются в Firebase Storage, а в Firestore хранятся ссылки на них, позволяя эффективно управлять изображениями и обеспечивая их доступность для клиентского приложения.

2.3 Серверная часть приложения

При проектировании классов и интерфейсов для серверной части приложения на базе Flutter (используя язык Dart) и Firestore, важно стремиться к минимальной избыточности и связности между компонентами, а также обеспечивать корректность кода, поскольку это существенно влияет на работу всех подключенных клиентов. Для достижения этой цели можно использовать хорошо продуманное распределение логики сервера по разным уровням, определять формат объектов, используемых на каждом уровне, и выполнять преобразования объектов при их передаче между уровнями. Также полезно применять соответствующие паттерны проектирования.

В данном случае выбран архитектурный стиль REST для обмена данными между клиентами и сервисами. Он широко использует возможности протокола HTTP и, следовательно, является состоянием (stateless). REST является современным подходом, применяемым во многих веб-сервисах, и имеет множество реализаций различных функций на его основе, таких как аутентификация, выполнение операций CRUD и т. д. В REST-приложениях наиболее распространенным форматом передачи данных является JSON, поэтому предлагается использовать его для обмена данными между клиентами и сервером.

При разработке серверной части приложения на Flutter с использованием Firestore, следует учесть особенности Firestore и его API. Firestore - это гибкая, масштабируемая облачная база данных, обеспечивающая реальное время и синхронизацию данных между клиентами.

2.3.1 Уровни. API, Service, Repository

Приняв во внимание вышенаписанное, лучше всего разбить серверную логику на следующие уровни API, Service и Repository.

API-уровень. Это самый верхний уровень сервера, и на нём будут располагаться классы-контроллеры, методы которых будут описывать URL-адреса, на которые можно будет сделать вызов, и их характеристику. Также эти методы будут преобразовывать данные из формата, пригодного для сериализации, в формат, пригодный для сохранения данных в Firestore и работы с ними как с бизнес-объектами.

Service-уровень. На этом уровне находятся классы, ответственные за бизнес-логику, среди них как те, которые просто делегируют вызов к репозиторию, так и те, которые выполняют более сложную логику.

Repository-уровень. Этот уровень содержит классы-репозитории, которые используются для выполнения операций с Firestore. Методы этих классов могут обращаться к Firestore API для создания, чтения, обновления и удаления данных. Репозитории могут выполнять автоматически сгенерированные запросы к Firestore или использовать пользовательские запросы в зависимости от требований приложения.

Пример класса-контроллера приведен в приложении Б.

2.3.2 Объекты, использующиеся на различных уровнях, и их особенности

Использование различных типов объектов для предоставления данных является хорошей практикой при разработке любых типов программных продуктов, а при наличии разных уровней в пределах одного приложения, без такого подхода обойтись попросту невозможно. Для этого использовались два паттерна одновременно: DTO-объект и Immutable-объект. Следуя первому паттерну, мы будем проектировать классы, содержащие только базовые типы, либо те типы, которые легко сериализуются. Его преимущество фактически уже было названо: понятная и предсказуемая сериализация, то есть разработчик всегда будет знать, как экземпляр класса будет выглядеть после превращения его в JSON-строку. Второй паттерн было решено применить по причине как раз-таки тех ограничений, что накладывает первый паттерн. DTO-объекты содержат в себе данные исключительно для передачи, более с ними никакие операции совершать не рекомендуется. Вывод о том, что такой объект должен быть неизменяемым, очевиден, более того, по мере роста проекта это поможет защитить программиста от дальнейших ошибок.

Второй тип объекта, которые будет использоваться на сервере – Entity-объект. Entity-объекты представляют собой объекты, которые включают данные, хранящиеся в Firestore или полученные из него. Они могут быть использованы на всех уровнях серверной логики. API-уровень может преобразовывать DTO-объекты в Entity-объекты для передачи их на Service-уровень, например, при обработке POST-запросов, а также преобразовывать Entity-объекты в DTO-объекты при запросе данных. Service-уровень использует Entity-объекты для выполнения бизнес-логики и взаимодействия с Repository-уровнем. Repository-уровень отвечает за операции с данными Firestore и может работать с Entity-объектами.

При работе с различными типами объектов и их взаимодействии важно определить, как будет осуществляться конвертация между ними. Для этого можно использовать специальные классы, называемые "мапперами" или "конвертерами", которые содержат логику преобразования данных из одного типа объекта в другой. В Flutter (Dart) с Firestore существует выбор библиотек, которые помогают автоматизировать и упростить этот процесс конвертации данных между различными объектами.

2.3.3 Безопасность. Аутентификация и авторизация

Важнейшим аспектом проектирования серверной части приложения является то, как пользователи будут входить в систему и как система будет понимать, что очередной присланный запрос прислан уже находящимся в системе пользователем.

Firebase Authentication предоставляет набор инструментов для аутентификации пользователей, включая поддержку различных провайдеров, таких как Google, Facebook, Twitter и другие. Он также позволяет создавать пользователей с помощью адреса электронной почты и пароля. Firebase Authentication обеспечивает безопасность ваших пользователей, управление сеансами пользователей и позволяет легко интегрировать аутентификацию в ваши клиентские приложения.

После того, как пользователь успешно пройдет процесс аутентификации с помощью Firebase Authentication и Google Auth, серверная часть может использовать JWT-токен (JSON Web Token) для поддержания состояния пользователя. JWT-токен представляет собой компактную и самодостаточную форму представления информации об аутентифицированном пользователе. Он содержит зашифрованную информацию о пользователе, такую как идентификатор пользователя, разрешения и срок действия токена.

Когда пользователь выполняет запрос к серверу, он должен предоставить JWT-токен в заголовке запроса или в другом безопасном месте. Серверная часть может проверить подлинность токена и его срок действия, а затем разрешить или отклонить запрос в зависимости от результата проверки. Это позволяет поддерживать состояние пользователя без хранения его данных на сервере и обеспечивает безопасность запросов.

2.3.4 Хранение фото

Реализовать функционал хранения фотографий можно как и вручную, храня их на собственном сервере, так и воспользовавшись уже готовым решением в облаке, такими как Firestore Database. В рамках работы над данным курсовым проектом, был выбран данный вариант, так как он предполагает наименьшие трудозатраты, однако соответствующие классы на серверной части приложения сделаны таким образом, что хранилище фотографий можно будет при необходимости легко сменить с облачное на удалённое либо локальное.

2.4 Клиентская часть приложения

Клиентская часть приложения не менее важна, чем серверная. Это то, с чем будет контактировать пользователь, то, от чего в первую очередь зависит будут ли впечатления от использования положительными или отрицательными. Здесь стоит понимать, исходя из специфики проекта, в первую очередь разрабатывается мобильное приложение и самые строгие требования со стороны пользователя будут предъявляться именно к нему.

Точно так же, как и серверная часть приложения, как и практически любой другой проект, который имеет риск сильно разрастись в будущем, лучше всего разбить логику клиентской части приложения на части таким образом, чтобы на этой основе можно было каждую часть реализовывать последовательно и независимо от других частей. Приняв во внимание все требования к функционалу, а также лучшие практики и паттерны, использующиеся при разработке Android-приложений, логику клиентской части приложения можно разделить на две категории: включающую в себя взаимодействие с пользовательским интерфейсом и ту, которая связана с бизнес-логикой.

В первую категорию будут входить:

* виджеты и разметка экранов;
* виджеты и разметка диалогов;
* адаптеры списков;
* стили, иконки.

Во вторую:

* классы, ответственные за внедрение зависимостей;
* функции, отправляющие запросы к API;
* прочие дополнительные и вспомогательные классы.

Понятно, что иерархически классы из категории с пользовательским интерфейсом стоят ниже, чем классы из категории с бизнес-логикой, и именно они будут пользоваться классами из второй категории.

2.4.1 Виджеты, управление состоянием виджетов, декларативный UI, фрагменты и диалоги, навигация

Так как было принято решение разрабатывать приложение на Flutter, были изучены различные способы управления состояниями виджетов, так как все объекты во флаттере являются виджетами, которые изменяют свое состояние и перерисовываются во время работы приложения. В больших проектах есть смысл в использовании такого паттерна проектирования как BLoC, который является посредником между уровнем представления и базой данных. В данном приложении было принято решение использовать метод setState() для контроля состояний.

Виджетами является всё то, что связано с интерфейсом: код, реагирующий на нажатия кнопок и на остальные события, код инициализации состояния, очистки состояния, валидация. Они по большей части выполняют работу по отображению пользователю какого-либо динамического контента, путём взаимодействия с соответствующей ему моделью, а также в ответ на действия пользователя. Важно отметить, что приложение будет построено целиком с использование Stateless и Statefull виджетах. Это стандартная практика при разработке приложений на Flutter, так как такая архитектура позволяет легко использовать одни и те же виджеты по нескольку раз, появляется возможность применять продвинутые библиотеки с навигацией и манипулировать их жизненным циклом.

Разметка в разработке на Flutter представляет собой декларативный UI, в которых с помощью классов виджетов описываются различные элементы интерфейса. Все сложные виджеты строятся на основе простых путем наследования одного от другого.

Навигация – очень важный для мобильного приложения элемент, при её проектировании необходимо поддерживать логическую связь между виджетами, давать возможность перейти на какой-либо виджет из одних виджетов и не позволять это делать из других. Необходимый список виджетов будет таким:

* с главным экраном, который будет содержать навигационное меню;
* со списком производителей;
* со списком товаров;
* с любимыми товарами;
* с корзиной пользователя;
* с профилем;
* с детальным описанием продукта;
* и некоторое количество диалогов, которые, по сути, тоже являются виджетами и участвуют в навигации наравне с ними.

Для управления навигацией в приложении используется пакет Get, который предоставляет удобные инструменты для навигации между виджетами. Он предоставляет более простой и декларативный подход к навигации, чем стандартный класс Navigator. С его помощью можно будет также строить граф переходов, где будут отображены стрелками направления переходов.

2.4.2 Внедрение зависимостей

Одной из самых распространённых практик при разработке мобильных приложений – это использование различных библиотек, внедряющихся виде зависимостей в pubsec файл. Для Flutter существует множество библиотек, предоставляющих различный функционал.

Во время разработки приложения были использованы такие библиотеки как:

* firebase\_core, firebase\_auth, cloud\_firestore и firebase\_storage, необходимые для выполнения запросов от приложения к Firebase;
* connectivity\_plus, для проверки соединения с Firestore;

2.4.3 Логика использования приложения

Как уже было написано ранее, типов пользователей всего два: пользователь и администратор. Как только регистрируется новый клиент, ему даются права обычного пользователя. Имея такие права, клиент может просматривать продукты, добавлять их в избранное и в корзину, а также просматривать профиль.

Пользователь заходит в приложение, изучив продукты он может добавить понравившееся в избранное, которое помечается закрашенной иконкой.

Администратор в свою очередь имеет возможность добавлять и изменять информацию, а также просматривать список заказов в веб-приложении.

2.4.4 Дизайн и расположение элементов

В виду своей кроссплатформенности, во Flutter присутствуют элементы Material Design для устройств, работающих по ОС Android и Cupertino для IOS, так как все приложения в App Store должны иметь схожий дизайн и не путать пользователей.

Приложение имеет в себе 4 главных экрана, являющиеся отправными точками для переходов на все остальные виджеты: экран с категориями, продуктами, брендами и с профилем. Для того, чтобы перемещаться между ними, используется ряд с кнопками в нижней части экрана, сделано это по той причине, что на рынке сейчас доминируют телефоны с большой диагональю, и дотянуться до нижней части экрана всегда проще всего. Перемещение по фрагментам назад осуществляется либо с помощью аппаратной кнопки назад, либо с помощью его программного аналога в левом верхнем углу.

При выборе цветовой гаммы приложения и вида интерфейса важно помнить для какой аудитории делается приложения и какие цели те или иные цвета и геометрические формы преследуют. Основной цвет приложения изменяется в зависимости от примененной пользователем системной расцветки.

В приложении используются различного вида кнопки, как имеющие заливку, так и нет. Первый тип используется для тех кнопок, которые делают какие-то существенные действия: добавление продуктов, новостей, выполняют вход в приложение и т.д. Второй же тип используется для менее важных действий, например, добавления товара в избранное или переход на новый экран. Такое разделение позволяет заострить внимание пользователя на важных действиях и не распылять его внимание на менее существенные.

Наличие виджетов, которые содержат в себе несколько типов полей ввода, а также имею несколько способов предоставлять пользователю информации, предоставляет дополнительные сложности при проектировании их дизайна, и при работе над ними нужно быть особенно внимательным, их нужно составлять таким образом, чтобы уместить достаточно количество информации на экран мобильного телефона, при этом не запутав пользователя. Для решения такой проблемы чаще всего используется группировка элементов, она может осуществляться, например, с помощью использования разделителей. Следуя последним трендам в дизайне, решено было использовать разделение контента на своеобразные «карточки» с закруглёнными углами. При этом стоит учитывать, что более натурально будет выглядеть, если карточка сверху будет иметь закруглённые углы только внизу, карточка посередине – и внизу, и вверху, а карточка внизу – только вверху.

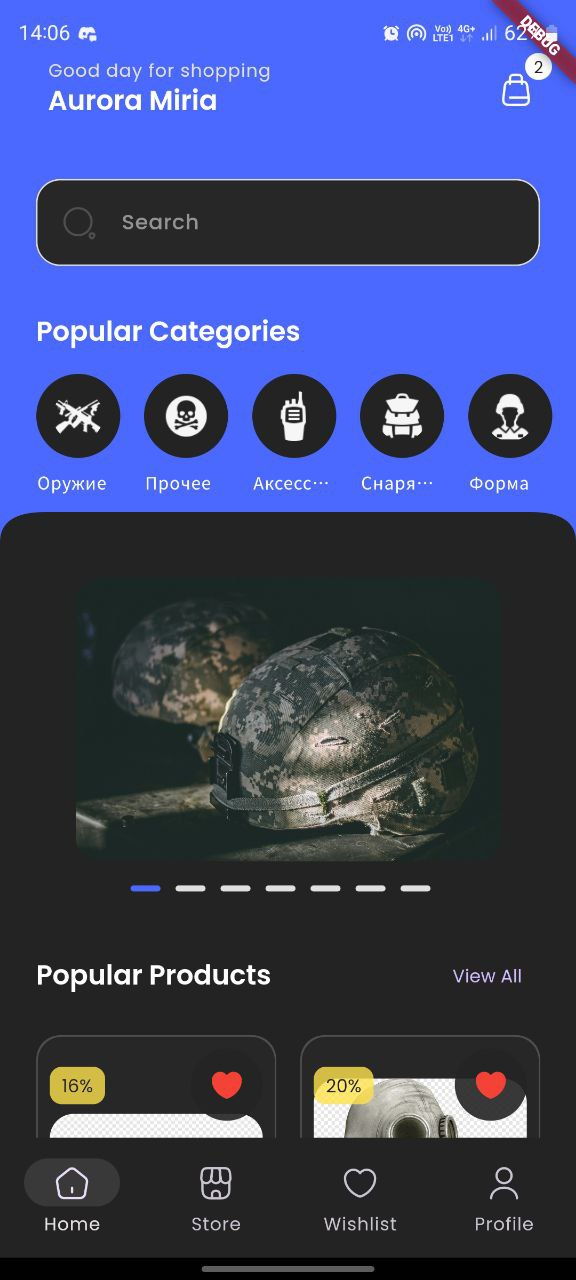
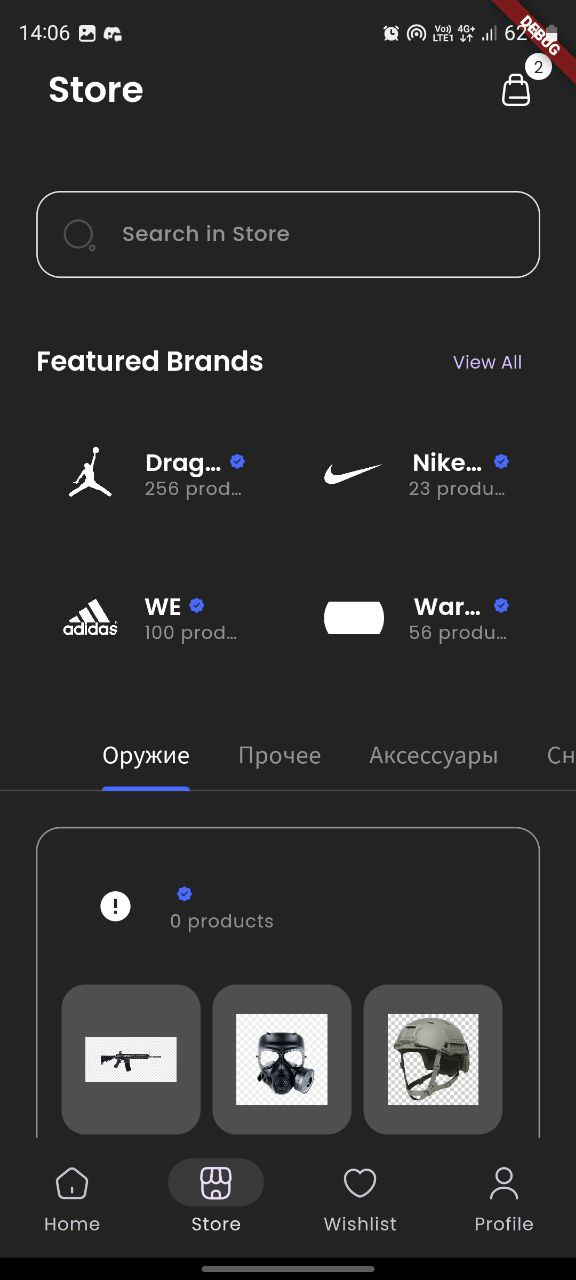
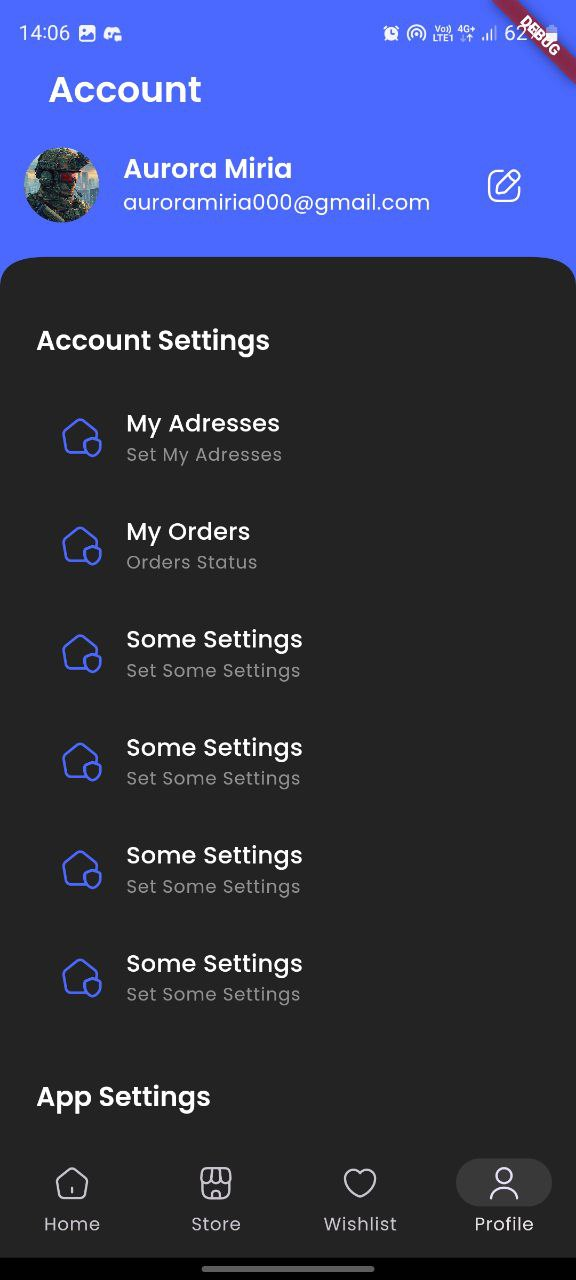
  

Рисунок 2.3 – Три основных экрана

2.5 Веб-приложение

В данной курсовой работе веб-приложение занимает самую малую часть. Оно целиком предназначено для пользователя с ролью «Администратор» и будет представлять собой страницу со входом, а после успешного входа – таблицу со списком салонов, где у него будет возможность менять их некоторые данные. Для реализации веб-приложения будет использоваться Vanilla JS.

2.6 Итоги раздела

По итогу данного раздела, у нас имеется детально проработанный план разработки приложения, как серверной и клиентской части, так и веб-приложения. Проработана внутренняя архитектура самих частей приложения, проработаны вопросы того, какого вида классы там будут использоваться, а также вопросы связи всех частей между собой и вопросы, связанные с безопасностью. Проект готов к реализации.

3 Реализация приложения

3.1 Серверная часть

Для разработки серверной части приложения был использован Firebase, платформа, предоставляемая Google, которая обеспечивает облачные услуги для разработки и развертывания приложений. Firebase предоставляет широкий набор инструментов и сервисов, которые упрощают разработку серверной части приложения и управление базой данных.

3.1.1 Конфигурация

Конфигурация серверной части приложения с использованием Firestore осуществляется через несколько ключевых компонентов и настроек.

Необходимо установить и настроить Firebase Admin SDK. Этот SDK предоставляет API для работы с различными сервисами Firebase, включая Firestore. Инициализация и аутентификация Firebase Admin SDK выполняются в начале серверного приложения с использованием файла ключа службы Firebase, полученного из Firebase Console. Этот файл содержит секретные ключи для аутентификации сервера и доступа к Firestore.

После инициализации Firebase Admin SDK следует настроить подключение к Firestore. Конфигурационные параметры, такие как адрес сервера Firestore, учетные данные аутентификации и другие настройки, могут быть указаны в файле .env или другом безопасном месте, чтобы защитить конфиденциальные данные.

Отдельно стоит уделить внимание форматам даты и времени, которые будут использоваться при сериализации. Это важный момент, так как очевидно, что тип, представляющий время и дату, является сложным, а значит, что на разных платформах, в разных языках программирования, этот тип может быть интерпретирован совершенно по-разному, поэтому возникает необходимость введения определённого единого стиля. Таковым был выбран стандарт ISO 8601, так как он является самым распространённым и имеет поддержку во всех используемых языках программирования. Он выглядит так: «YYYY-MM-DDThh:mm:ss».

Вся эта конфигурация позволит взаимодействовать с Firestore из вашего серверного приложения и выполнять операции чтения и записи данных в базу данных Firestore.

3.1.2 Реализация REST API. Контроллеры

Классы-контроллеры в приложении, использующем Firestore, представляют API. У них есть методы, которые вызываются при поступлении HTTP-запросов на сервер. Шаблон описания каждого контроллера включает декораторы, которые указывают его характеристики. В данном случае, мы будем использовать декоратор, чтобы указать, что это REST-контроллер. Внутри класса контроллера определяются методы, каждый из которых помечен аннотацией, указывающей тип HTTP-метода, данные, принимаемые методом, и данные, возвращаемые им.

При использовании Firestore с помощью Firebase Admin SDK, валидация входных данных может быть реализована с помощью встроенных механизмов Firestore. Firestore позволяет определить правила доступа и ограничения на структуру данных в базе. Это позволяет автоматически проверять входные данные на соответствие заданным правилам и возвращать ошибку, если данные не проходят валидацию.

Кроме того, в классы-контроллеры могут быть внедрены сервисы для работы с Firestore. Это позволяет контроллерам использовать сервисы для выполнения операций чтения и записи данных в Firestore. Сервисы могут быть созданы и настроены отдельно, а затем внедрены в контроллеры с помощью механизма Dependency Injection.

Таким образом, классы-контроллеры играют важную роль в организации API и взаимодействии с Firestore, обеспечивая обработку HTTP-запросов и использование сервисов для работы с данными в базе Firestore.

3.1.3 Сервисы.

Сервисы – классы, которым контроллеры делегируют выполнение различных действий, после поступления на них запроса. Они также помечаются своим особым декоратором, но скорее для того, чтобы фреймворк смог создать по ним объекты и включить их в свой пул для последующего внедрения зависимостей.

Как было уже описано ранее, большинство сервисов лишь выполняют различные действия с репозиториями, иногда лишь делая небольшие преобразования в объектах, чтобы ORM-система смогла правильно их обработать. Работа сервисов совершается в неблокирующем асинхронном режиме. В конструктор сервиса так же встраиваются различные модели.

3.1.4 Репозитории. Взаимодействие с Firestore

Для взаимодействия с базой данных Firestore вместо ORM sequelize-typescript используется Firebase Admin SDK и Firestore API. В Firestore данные организованы в коллекции, которые могут содержать документы. Каждый документ представляет собой набор полей со значениями. Для этого создаются Модели – классы, представляющие собой коллекции в Firestore. Эти классы принимают на вход методы репозиториев. Репозиторий в нашем случае – это сами классы моделей, которые позволяют работать с ними как с репозиториями.

При создании коллекции и документа в Firestore, нет необходимости явно определять схему или ограничения для полей. Firestore является гибкой NoSQL базой данных, в которой каждый документ может иметь свою структуру и поля. Ограничения и валидацию данных можно выполнять на уровне приложения, перед отправкой данных в Firestore.

Пример репозитория приведен в приложении А.

3.1.5 DTO. Конвертация разных типов объектов

DTO – важнейшая часть API-системы, так как эти объекты представляют то, как должны будут выглядеть запросы и ответы к ней. Легко понять тот факт, что если запросы будут по большей части одинаковыми, то ответы могут отличаться как по своему наполнению, так и по структуре. Например, если приходит запрос на получение одного пользователя, то не имеет смысла отправлять вместе с ним полную информацию о всех консультациях, в которых он поучаствовал, так как большинство из отправленных данных не будет использоваться получателем и канал передачи будет впустую нагружаться. Также будут возникать проблемы как с многочисленными зависимостям, если они будут тянуть друг друга, так и с циклическими зависимостям, которые будут вызывать переполнение стека. Одним из решений перечисленных проблем является определение класса для каждого из типов запросов. Главным его недостатком является то, что придётся писать достаточно большое количество мало чем отличающегося кода. Тем не менее, преимуществ такой подход несёт больше, поэтому был выбран именно он. Для реализации этого подхода использовались:

* классы для каждого тела запроса;
* классы, которые наследовали либо все поля либо часть из них.

Пример такого класса можно увидеть в приложении Б.

При валидации используется npm-пакет class-validator.

Так же в приложении реализована загрузка тела с Content-Type двух видов multipart/form-data и application/json. Так как при загрузке информации с телом переданным как multipart/form-data вся информация приходит в виде строки, испольщовался пакет class-transorm для преобразования типа строк в тип указынный в DTO с помощью декораторов @Transform

3.1.6 Аутентификация и авторизация

Firebase предоставляет встроенные функции аутентификации и авторизации, которые облегчают процесс работы с пользователями. Для аутентификации пользователей используется Firebase Authentication, а для авторизации — Firebase Firestore и его правила доступа.

При использовании Firebase Authentication, пользователи могут войти в приложение с помощью учетных записей Google, используя Google Auth. Firebase обрабатывает процесс аутентификации и генерирует уникальный идентификатор пользователя (UID), который можно использовать для идентификации и управления доступом к данным.

Вместо JWT-токенов, Firebase Authentication использует специальные токены Firebase, которые автоматически генерируются и проверяются серверной частью Firebase при выполнении запросов к API. Токены Firebase содержат информацию о текущем аутентифицированном пользователе и его разрешениях.

Для авторизации пользователей и управления доступом к данным в Firestore используются правила доступа (Firestore Security Rules). Правила доступа позволяют определить, какие пользователи имеют доступ к определенным коллекциям и документам в Firestore, основываясь на их UID или других атрибутах.

Таким образом, вместо пакета passport и JWT-токенов, в приложении на основе Firebase и Google Auth используются встроенные функции Firebase Authentication, Google Auth и Firestore Security Rules для обеспечения аутентификации и авторизации пользователей.

3.2 Клиентская часть

Разработка приложений под Android имеет свою специфику и во многом отличается от разработки для других платформ. Изначально в SDK идут только самые базовые компоненты, которые очень сложно использовать для того, чтобы реализовать требуемый функционал, однако существует большое множество библиотек и фреймворков, предоставляемых в том числе и разработчиками Google.

Самые главные сторонних компонентов, которые использовались, это:

* flutter\_http – библиотека, упрощающая совершение вызовов к API;
* Future– библиотека для выполнения асинхронного кода;
* Material и Cupertino – библиотеки, предоставляющая компоненты из Material UI и Cupertino.

Языком программирования был выбран Dart.

3.2.1 Виджеты, разметка, диалоги.

Так как в проекте используется DataBinding, то при создании вида каждого фрагмента приходится использовать специальные классы, которые генерируются этим фреймворком. Для этого, а также для того, чтобы сократить количество остального повторяющегося кода, был создан абстрактный класс StandardBoundFragment. Туда, в качестве generic-параметров, посылается класс ViewModel и класс, который обеспечивает привязку данных. Также в качестве аргументов конструктора туда посылаются методы создания класса с привязками, связывания ViewModel, внедрения зависимостей и уничтожения связанного компонента. Далее в нужных местах эти методы вызываются, а также инициализируются поля, которые, как показала практика, будут использоваться почти в каждом фрагменте: поле с классом связывания и поле с ViewModel. Также этот класс предоставляет два метода для реализации: initElements и subscribeUi. Первый нужен для того, чтобы инициализировать UI-компоненты, а второй, чтобы подписаться на обновления полей из ViewModel.

Диалоги, разработанные в данном проекте, тоже являются виджетами. Они используются при вызове функции удаления элемента для того чтобы подтвердить операцию. Диалоговое окно представляет собой объект класса AlertDialog, который включает виджет Text – для отображения текста сообщения и виджеты MaterialButton – для расположения в нес кнопок отмены и подтверждения. При нажатии кнопку подтвердить вызывается функция delete, в теле которой происходит отправка запроса на удаление. После удаления сервер возвращает удаленный объект и при помощи метода setState() происходит перерисовка виджета.

3.2.2 Внедрение зависимостей

При разработке мобильной части приложения повсеместно используется внедрение зависимостей. Помимо того, что оно предоставляет единое место для создания объектов, он также предоставляет возможности взаимодействия с жизненным циклом фрагментов, что крайне полезно в случае, если необходимо иметь ViewModel, использующуюся более чем в одном фрагменте, например, когда создаётся новое соревнование. Классы для внедрения зависимостей расположены в специально созданной для этого директории и могут разделяться на три типа: компоненты, модули и области видимости. Помимо этого, используется собственный класс, который называется ComponentKeeper, он содержит в себе все компоненты, обёрнутые в класс DestroyableComponent. Всё это нужно для того, чтобы можно было управлять жизненным циклом компонентов, уничтожая их в нужный момент (например, при выходе из фрагмента). Области видимости же нужны для того, чтобы Dagger понимал, когда используется новый компонент, а когда тот же самый. Каждому фрагменту соответствует свой компонент, при этом все они объединены в один большой, принадлежащий всему приложению, который создаётся в классе MyApplication, и через который создаются все остальные компоненты.

3.3 Итоги раздела

В итоге была разработана мобильная и серверная часть приложения, а также проверено и отлажено их взаимодействие. Основная сложность была в грамотной реализации поддержки соревновательных сессий, чтобы одни клиенты исправно отправляли координаты на сервер, а другие клиенты их получали, а также в том, чтобы правильно реализовать обработку этих локаций. Также много сложностей возникало с правильным структурированием всех взаимодействующих классов в мобильном приложении, что было, несмотря на это, успешно выполнено.

4 Тестирование, проверка работоспособности и анализ полученных результатов

Тестирование проекта выполнялось в течение разработки посредством таких средств, как отладка и логгирование. После введения нового либо изменения существующего компонента программы, он и зависящий от него код также подвергался тестированию. Анализ полученных результатов происходил по факту получения ошибок и проблемные места сразу исправлялись. После завершения разработки программы по нескольку раз были подвергнуты испытаниям все элементы управления в пользовательском интерфейсе, также они тестировались по ходу написания записки. Если происходили ошибки, то их причина и местоположение в коде легко фиксировались благодаря ведению логов.

После написания приложения также проводились различные тесты, и проверки, такие как проверка на корректное завершение соревнования, проверка на корректную работу таймера, проверка корректного создания соревнования и команды, а также их удаления и изменения.

В итоге приложение стало хорошо отлаженным и готовым к полноценному использованию.

5 Руководство по использованию

Для того, чтобы установить приложение на телефон, Вам необходимо иметь версию Android 7.0 и выше (версию API 22 и более). Версию Вашего телефона можно узнать в настройках.

Если Ваша версия удовлетворяет требованиям, то можно устанавливать приложение на телефон.

После установки приложения на экране появится иконка и подпись “strikeshop” (рис.5.1)

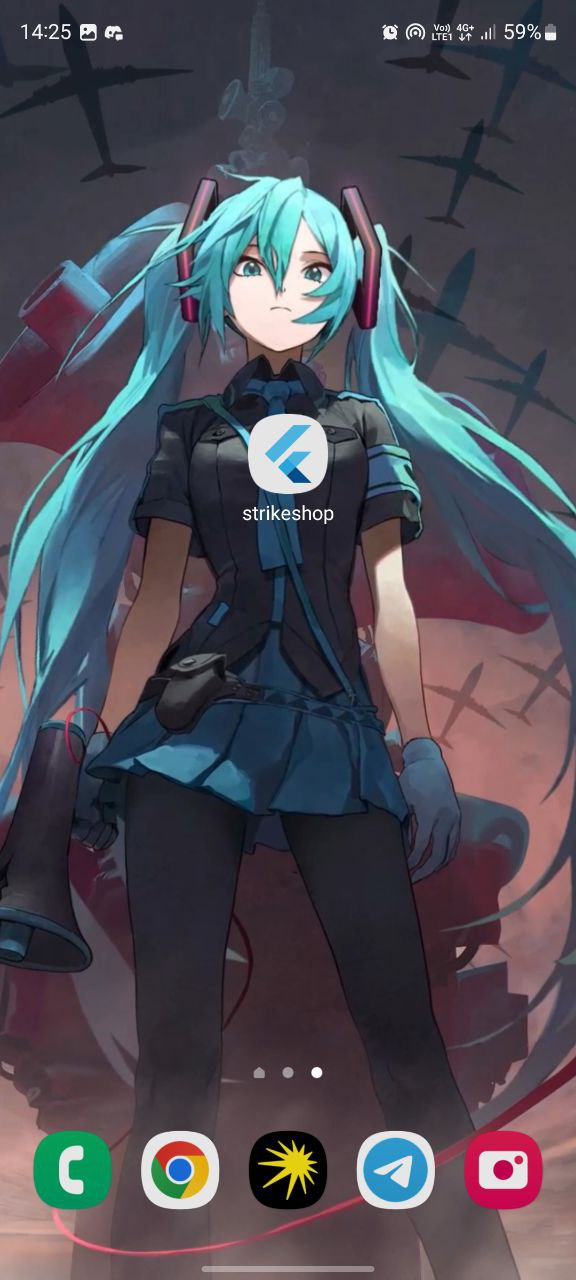


Рис 5.1 Иконка приложения

Изначально открывается экран, на котором находится форма авторизации, которая имеет заголовок, два текстовых поля и две кнопки: регистрации и авторизации(рис.5.2).

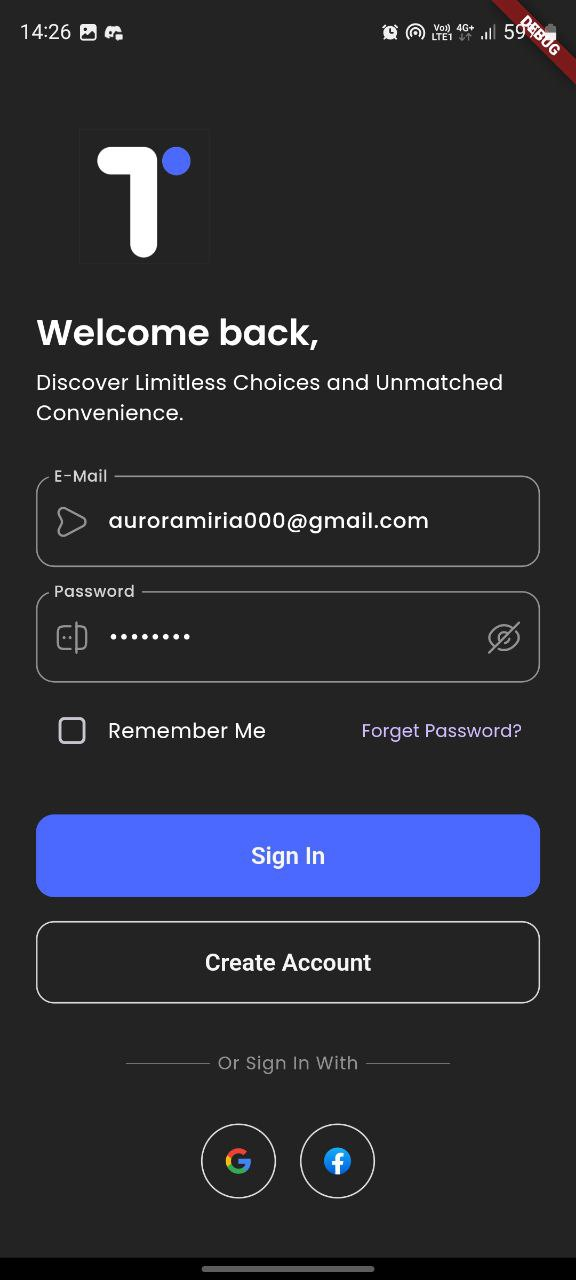


Рисунок 5.2 – Форма авторизации

Перед первым использованием приложения Вам необходимо зарегистрироваться. Для этого нужно заполнить поля логин и пароль валидными данными и нажать на кнопку “Create Account”.

Вы успешно прошли регистрацию и можете зайти в свой аккаунт. Для этого Вы заполняете email (имя пользователя) и вводите пароль (password) на форме входа, после нажатия клавиши “SING IN” Вы переходите в приложение.

Сразу открывается главный экран, где отображаются популярные товары (рис.5.3).

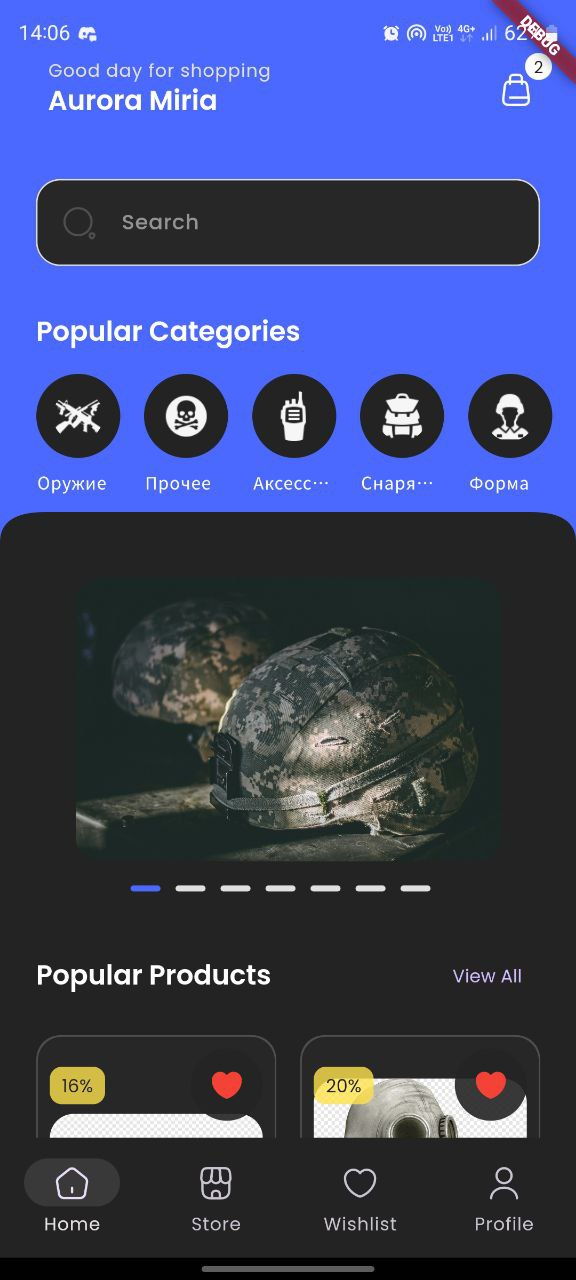


Рисунок 5.3 – Главная страница

Все элементы списка оформлены в виде карточек и при нажатии на определенный элемент списка будет открыт экран с подробным описанием продукта. Также на данном экране можно добавить товар в избранное кликнув на иконку сердечка и удалить из избранного повторным нажатием. Перед удалением появляется диалоговое окно с подтверждением удаления, что так же позволит избежать ошибки пользователя. (рис. 5.4).

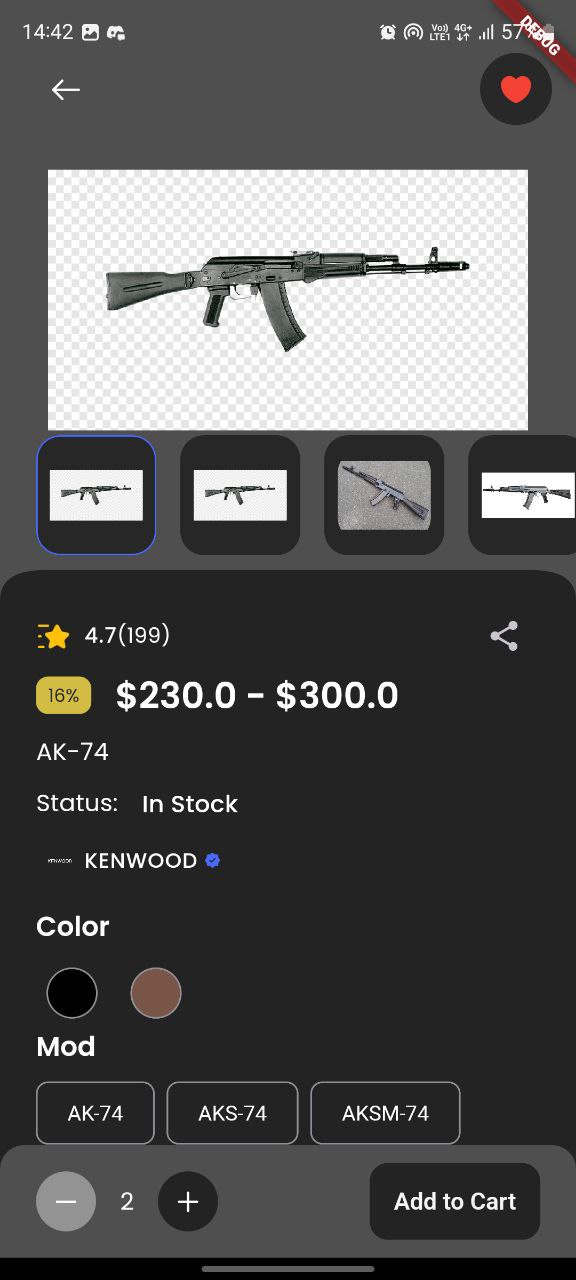


Рисунок 5.4 – Страница просмотра товара

Последним экраном является профиль, на котором просто выводиться информация об авторизованном пользователе.

Заключение

При работе над курсовым проектом было проведено изучение предметной области, выявление в ней проблем и составление функционала программного средства, с их учётом, после чего последовало проектирование и разработка. Также в ходе работы проанализировано и проработано множество сторон и нюансов проектирования и разработки мобильного приложения, сервера, а также веб-приложения, например, выбор подходящих паттернов и технологий, составление и тестирование алгоритмов, структурирование проекта, построение пользовательского интерфейса, синхронизации данных и т.д.

В итоге были реализован весь заявленный функционал и готовое приложение вполне можно назвать подходящим для реального использования. Как мобильная, так и серверная части проекта имеют хороший потенциал для будущих модификаций во многом благодаря грамотному структурированию, внимательностью при написании кода, а также выборе известных и проверенных сопутствующих технологий. Было получено большое количество опыта по работе с такими фреймворками как Flutter, а также Firebase, улучшены навыки по проектированию, а также применению и связыванию друг с другом различных технологий.

Список литературы

1. REST-API [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/483202/>. – Дата доступа: 16.05.2024.
2. StudFiles [Электронный ресурс]. – Режим доступа: https://studfiles.net/preview/page:2/. – Дата доступа: 16.05.2024.
3. Трехуровневая архитектура — Википедия [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/Трехуровневая\_архитектура. – Дата доступа: 16.05.2024.
4. Документация по Android [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://developers.google.com/android/> – Дата доступа: 16.05.2024;
5. Мартин, Р. Чистая архитектура и искусство разработки программного обеспечения / Р. Мартин. – Санкт-Петербург: Питер, 2019. – 352 с
6. Metanit Java [Электронный ресурс]. – Режим доступа: <https://metanit.com/java/android/>. – Дата доступа: 16.05.2024
7. Голощапов А. Google Android: программирование для мобильных устройств / Голощапов А. — СПб.: БХВ-Петербург, 2010. — 4

Приложение А

import 'dart:io';

import 'package:cloud\_firestore/cloud\_firestore.dart';

import 'package:flutter/services.dart';

import 'package:get/get.dart';

import 'package:strikeshop/data/services/firebase\_cloud\_storage.dart';

import 'package:strikeshop/features/shop/models/product\_model.dart';

import 'package:strikeshop/utils/constants/enums.dart';

import 'package:strikeshop/utils/exceptions/firebase\_exceptions.dart';

import 'package:strikeshop/utils/exceptions/platform\_exceptions.dart';

class ProductRepository extends GetxController {

static ProductRepository get instance => Get.find();

final \_db = FirebaseFirestore.instance;

Future<List<ProductModel>> getFeaturedProducts() async {

try {

final snapshot = await \_db

.collection('Products')

.where('IsFeatured', isEqualTo: true)

.limit(4)

.get();

return snapshot.docs.map((e) => ProductModel.fromSnapshot(e)).toList();

} on FirebaseException catch (e) {

throw SFirebaseException(e.code).message;

} on PlatformException catch (e) {

throw SPlatformException(e.code).message;

} catch (e) {

throw 'Something went wrong. Please try again.';

}

}

Future<List<ProductModel>> getAllFeaturedProducts() async {

try {

final snapshot = await \_db

.collection('Products')

.where('IsFeatured', isEqualTo: true)

.get();

return snapshot.docs.map((e) => ProductModel.fromSnapshot(e)).toList();

} on FirebaseException catch (e) {

throw SFirebaseException(e.code).message;

} on PlatformException catch (e) {

throw SPlatformException(e.code).message;

} catch (e) {

throw 'Something went wrong. Please try again.';

}

}

Future<List<ProductModel>> fetchProductsByQuery(Query query) async {

try {

final querySnapshot = await query.get();

final List<ProductModel> productList = querySnapshot.docs

.map((doc) => ProductModel.fromQuerySnapshot(doc))

.toList();

return productList;

} on FirebaseException catch (e) {

throw SFirebaseException(e.code).message;

} on PlatformException catch (e) {

throw SPlatformException(e.code).message;

} catch (e) {

throw 'Something went wrong. Please try again.';

}

}

Future<void> uploadProductData(List<ProductModel> products) async {

try {

final storage = Get.put(SFirebaseStorageService());

for (var product in products) {

final thumbnail =

await storage.getImageDataFromAssets(product.thumbnail);

final url = await storage.uploadImageData(

'Products/Images', thumbnail, product.thumbnail.toString());

product.thumbnail = url;

if (product.images != null && product.images!.isNotEmpty) {

List<String> imagesUrl = [];

for (var image in product.images!) {

final assetImage = await storage.getImageDataFromAssets(image);

final url = await storage.uploadImageData(

'Products/Images', assetImage, image);

imagesUrl.add(url);

}

product.images!.clear();

product.images!.addAll(imagesUrl);

}

if (product.brand != null) {

final brandImage =

await storage.getImageDataFromAssets(product.brand!.image);

final brandImageUrl = await storage.uploadImageData(

'Brands', brandImage, product.brand!.name);

product.brand!.image = brandImageUrl;

}

if (product.productType == ProductType.variable.toString()) {

for (var variation in product.productVariations!) {

final assetImage =

await storage.getImageDataFromAssets(variation.image);

final url = await storage.uploadImageData(

'Products/Images', assetImage, variation.image);

variation.image = url;

}

}

await \_db.collection("Products").doc(product.id).set(product.toJson());

}

} on FirebaseException catch (e) {

throw e.message!;

} on SocketException catch (e) {

throw e.message;

} on PlatformException catch (e) {

throw e.message!;

} catch (e) {

throw e.toString();

}

}

}

Приложение Б

import 'package:get/get.dart';

import 'package:strikeshop/data/repositories/products/product\_repository.dart';

import 'package:strikeshop/features/shop/models/product\_model.dart';

import 'package:strikeshop/utils/constants/enums.dart';

import 'package:strikeshop/utils/popups/loaders.dart';

class ProductController extends GetxController {

static ProductController get instance => Get.find();

final isLoading = false.obs;

final productRepository = Get.put(ProductRepository());

RxList<ProductModel> featuredProducts = <ProductModel>[].obs;

@override

void onInit() {

fetchFeaturedProducts();

super.onInit();

}

void fetchFeaturedProducts() async {

try {

isLoading.value = true;

//Fetch Products

final products = await productRepository.getFeaturedProducts();

//Assign Products

featuredProducts.assignAll(products);

} catch (e) {

SLoaders.errorSnackBar(title: 'Error', message: e.toString());

} finally {

isLoading.value = false;

}

}

Future<List<ProductModel>> fetchAllFeaturedProducts() async {

try {

//Fetch Products

final products = await productRepository.getAllFeaturedProducts();

return products;

} catch (e) {

SLoaders.errorSnackBar(title: 'Error', message: e.toString());

return [];

}

}

String getProductPrice(ProductModel product) {

double smallestPrice = double.infinity;

double largestPrice = 0.0;

if (product.productType == ProductType.single.toString()) {

return (product.salePrice > 0 ? product.salePrice : product.price)

.toString();

} else {

for (var variation in product.productVariations!) {

double priceToConsider =

variation.salePrice > 0.0 ? variation.salePrice : variation.price;

if (priceToConsider < smallestPrice) {

smallestPrice = priceToConsider;

}

if (priceToConsider > largestPrice) {

largestPrice = priceToConsider;

}

}

if (smallestPrice.isEqual(largestPrice)) {

return largestPrice.toString();

} else {

return '$smallestPrice - \$$largestPrice';

}

}

}

String? calculateSalePercentage(double originalPrice, double? salePrice) {

if (salePrice == null || salePrice <= 0.0) {

return null;

}

if (originalPrice <= 0) {

return null;

}

return (((originalPrice - salePrice) / originalPrice) \* 100.0)

.toStringAsFixed(0);

}

String getProductStockStatus(int stock){

return stock > 0 ? 'In Stock' : 'Out of Stock';

}

}