

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский ядерный университет «МИФИ»
Обнинский институт атомной энергетики –
филиал федерального государственного автономного образовательного учреждения высшего
образования «Национальный исследовательский ядерный университет «МИФИ»
(ИАТЭ НИЯУ МИФИ)

Отделение Интеллектуальные кибернетические системы

**Выпускная квалификационная работа –
бакалаврская работа**

по направлению подготовки: 09.03.02 Информационные системы и технологии
Направленность (профиль): Информационные технологии

«Разработка приложения «Shopping Assistant» для операционной системы Android»
(название работы)

Выполнил:

студент гр. _____

_____ (подпись, дата)

Галиханов А.Ф.

Руководитель ВКР,
к.т.н., доцент ОИКС

_____ (подпись, дата)

Грицюк С.В.

Нормоконтроль

_____ (подпись, дата)

Пичугина И.А.

Выпускная квалификационная
работа допущена к защите

_____ (№ протокола, дата заседания комиссии)

Руководитель
образовательной программы
09.03.02 Информационные
системы и технологии
канд. тех. наук

_____ (подпись, дата)

Мирзеабасов О.А.

Обнинск, 2021 г

Реферат

Работа 43 с., 1 табл., 14 рис., 12 ист., 2 прил.

API, АРХИТЕКТУРА REST, СУБД, JSON , FLASK, HTTP, JSON Web Token, ANDROID, ПАТТЕРН MVVM.

Объектом исследования является архитектурный стиль REST и паттерн проектирования MVVM для реализации клиент-серверного приложения.

Цель работы – разработка клиент-серверного приложения «ShoppingAssistant».

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ	5
1.1 Исследование предметной области	5
1.2 Выявление функциональных и системных требований	6
1.3 Анализ существующих аналогов приложения	7
ГЛАВА 2. ОБЗОР ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ	9
2.1 Архитектура REST	9
2.2 Формат передачи данных JSON	11
2.3 Стандарт JSON Web Token	11
2.4 Шаблон проектирования MVVM	13
2.5 Операционная система android	14
2.6 Выбор и обоснование инструментов разработки	16
ГЛАВА 3. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ . . .	18
3.1 Проектирование базы данных	18
3.2 Реализация серверной части проекта	20
3.3 Реализация клиентской части проекта	26
ЗАКЛЮЧЕНИЕ	32
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	33
ПРИЛОЖЕНИЕ А	35
ПРИЛОЖЕНИЕ Б	39

ВВЕДЕНИЕ

Питание является одним из основных факторов, влияющих на здоровье человека. В настоящее время заметно возрастает понимание того, что пища оказывает на человека значительное влияние. Она даёт энергию, силу, развитие, а при грамотном её употреблении – и здоровье. Без всякого сомнения, можно утверждать, что здоровье человека на 65% зависит от питания. Вредная пища зачастую является основным источником большинства заболеваний. Повышенное содержание холестерина в крови, ожирение, кариес, диабет, нарушение жирового обмена веществ, гипертония, повышенное содержание мочевой кислоты в крови – вот неполный перечень болезней вызванных неправильным питанием. Исключив из своего рациона вредные продукты, можно не только предотвратить эти заболевания, но и избавиться от большинства хронических недугов. С каждым годом увеличивается ассортимент готовых продуктов питания и усложняется их состав. Вместе с тем растут и проблемы выбора здоровой пищи возле прилавков магазина. Ведь не всегда просто разобрать состав продукта, написанный мелким шрифтом, и определить подходит ли данный товар вам. Особенно эта проблема актуальна для людей со слабым зрением или имеющих аллергию на определенные продукты. Для облегчения процесса выбора продуктов питания, было принято решение, реализовать приложение «ShoppingAssistant»(Помощник по покупкам).

ShoppingAssistant ускоряет и облегчает процесс покупок в любых продовольственных магазинах. Просто отсканировав штрих-код продукта с помощью камеры на смартфоне, через приложение, получаешь полную информацию о составе и пищевой ценности продукта в удобочитаемом виде. В приложении можно заранее составлять список нежелательных продуктов в составе товара, которые будут подсвечиваться определенным цветом предупреждая вас. Также ведется подсчет калорий и соотношения белков, жиров и углеводов в корзине для соблюдения баланса энергетической ценности.

Целью данной работы является разработка клиент-серверного приложения «ShoppingAssistant», ориентированное на людей следящих за своим питанием:

- 1) Людей имеющих аллергию на определенные продукты;
- 2) Вегетарианцев;
- 3) Людей соблюдающих пост по религиозным или иным причинам;
- 4) Людей соблюдающих диету для похудения или для набора мышечной массы.

При релизации сервера придерживаться архитектурного стиля REST(Representational State Transfer — «передача состояния представления»). Для взаимодействия с клиентом сервер должен предоставлять API(Application Programming Interfaces — «интерфейс прикладного программирования»). Использовать данные полученные и подготовленные в рамках курса «Нереляционные базы данных». При реализации клиентского приложения использовать архитектурный паттерн MVVM.

ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Исследование предметной области

В ходе исследования выявлены следующие сущности характеризующие предметную область данной работы:

- 1) Пользователь – потенциальный клиент пользующийся услугами разрабатываемого приложения
- 2) Продукт – переработанный и произведенный продовольственный товар предназначенный для употребления человеком в пищу.
- 3) Недопустимый продукт – нежелательный к употреблению ингредиент для пользователя, входящий в состав продукта.

Целью пользователя является получение информации о продукте. Поэтому пользователю необходимо предоставить удобный инструмент для поиска продуктов в разрабатываемом сервисе. Для этого необходимо однозначно идентифицировать продукты в системе. Одним из возможных вариантов решения этой проблемы, является использование штрих-кода. Рассмотрим более подробно процесс маркировки товаров и выясним целесообразность использования штрих-кода в качестве уникального идентификатора продукта.

Каждый произведенный и допущенный к продаже продовольственный товар должен маркироваться штрих-кодом. Штрих-код это последовательность чёрных и белых полос, нанесенная на поверхность, маркировку или упаковку товаров, предоставляющая возможность считывания её техническими средствами. Двоичная система цифр обеспечивает удобную запись штрих кодов. Штрихи обозначаются цифрой «1», а пробелы «0». В штрих-код зашифровывается номер GTIN. GTIN обеспечивает однозначную идентификацию товара в любой стране мира и не может быть присвоен никакому иному товару. Функции администрирования номера GTIN в каждой стране возложена на национальную организацию GS1. GTIN имеет четкую

структуру — то есть строится по определенным правилам. В зависимости от назначения товара, ему может быть присвоено GTIN длиной 8, 12, 13 или 14 цифр. Большинство товаров во всем мире кодируют системой EAN-13. Код EAN-13 состоит из 13 цифр и имеет следующую структуру:

- 1) Первые 3 цифры кода EAN-13 обозначают страну производства товара;
- 2) Следующие 4 цифры являются кодом предприятия изготовителя товара;
- 3) Следующие 5 цифр – это код товара по классификации изготовителя;
- 4) 13-я цифра – контрольное число, которое вычисляется из предыдущих двенадцати.

Нанесение штрих-кода на товар осуществляет производитель с применением данных о стране местонахождения и кода производителя. Из вышесказанного следует, что штрих-код является лучшим выбором для идентификации продуктов в нашем сервисе. Использование такого подхода позволит реализовать сканер штрих-кода, что несомненно увеличит удобство использования приложения.

1.2 Выявление функциональных и системных требований

Реализуемый сервис должен удовлетворять следующим функциональным требованиям:

- 1) Регистрация пользователя. Все данные пользователя должны храниться в БД. Для обеспечения безопасности, пароль доступа должен храниться в зашифрованном виде;
- 2) Авторизация пользователя. Процесс аутентификации пользователя должна происходить с помощью токенов доступа. Сервер должен иметь возможность обновления токена;
- 3) У каждого зарегистрированного пользователя в базе данных, должен храниться список недопустимых продуктов. Пользователь должен иметь возможность добавлять и удалять необходимые продукты;
- 4) По штрих-коду товара сервер должен предоставить пользователю:
 - состав товара;

- пищевую и энергетическую ценность товара;
- список недопустимых продуктов входящих в состав товара.

Для реализуемого сервиса предъявляются следующие системные требования:

- 1) При проектировании сервера использовать архитектурный стиль REST;
- 2) Для передачи данных клиенту и взаимодействия с базой данных использовать формат JSON;
- 3) Для реализации системы авторизации использовать технологию и стандарт JWT(JSON Web Token). На сервере не должна храниться никакая информация о состоянии клиента;
- 4) Импортировать в базу данных разрабатываемого сервера данные, полученные и подготовленные в рамках курса «Нереляционные базы данных».
- 5) Клиент должен быть реализован для операционной системы Android с применением шаблона проектирования MVVM.

1.3 Анализ существующих аналогов приложения

В ходе поиска аналогов со схожим функционалом, были найдены два приложения «Open Food Facts» и «Состав продуктов». Мы провели небольшой анализ этих приложений, в ходе которого выявили достоинства и недостатки каждого из них.

«Open Food Facts» – это бесплатная онлайн-база данных о пищевых продуктах со всего мира, функционирующая под лицензией Open Database License (ODBL). Проект призван собирать информацию и данные о пищевых продуктах со всего мира. Для каждого продукта в базе данных хранится его общее название, тип упаковки, бренд, категория, места производства, страны и магазины, где продается продукт, список ингредиентов. Для взаимодействия с этой базой данных реализовано приложение, которое можно скачать в Google Play.

Достоинства:

- 1) Большая база данных с продуктами по всему миру.
- 2) Возможность просмотра полной информации о продукте
- 3) Поиск продукта по штрих-коду
- 4) Наличие сканера штрих-кода

Недостатки:

- 1) В базе данных представлено мало российских продуктов
- 2) Не удобное для чтения отображение информации о продукте
- 3) Отсутствует возможность добавления недопустимых для пользователя продуктов в составе товара.

«Состав продуктов» – приложение предоставляет информацию о питательной ценности продуктов и их химическом составе: содержание белков, жиров, углеводов, калорийности, витаминов, минеральных веществ.

Достоинства:

- 1) Большая база данных с российскими продуктами.
- 2) Возможность просмотра полной информации о продукте

Недостатки:

- 1) Отсутствует поиск по штрих коду продукта. Присутствует возможность выбора из предложенных категорий или поиска по названию. В отличие от штрих кода название не является уникальным свойством продукта и не может его полностью идентифицировать.
- 2) Отсутствует возможность добавления недопустимых для пользователя продуктов в составе товара.

В ходе проведенного анализа не найдены аналоги покрывающие весь функционал разрабатываемого приложения.

ГЛАВА 2. ОБЗОР ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ

2.1 Архитектура REST

Термин REST ввел Рой Филдинг, один из создателей протокола HTTP, в своей докторской диссертации "Архитектурные стили и дизайн сетевых программных архитектур" ("Architectural Styles and the Design of Network-based Software Architectures")[1] в 2000 году. REST — это акроним, сокращение от английского Representational State Transfer — передача состояния представления. Архитектурный стиль взаимодействия компонентов распределенной системы в компьютерной сети. REST определяет стиль взаимодействия (обмена данными) между разными компонентами системы, каждая из которых может физически располагаться в разных местах. Данный архитектурный стиль представляет собой согласованный набор ограничений, учитываемых при проектировании распределенной системы. Эти ограничения иногда называют принципами REST:

- 1) Приведение архитектуры к модели клиент-сервер. В основе данного ограничения лежит разграничение потребностей. Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Данное ограничение повышает переносимость клиентского кода на другие платформы, а упрощение серверной части улучшает масштабируемость системы. Само разграничение на "клиент" и "сервер" позволяет им развиваться независимо друг от друга;
- 2) Отсутствие состояния. Архитектура REST требует соблюдения следующего условия. В период между запросами серверу не нужно хранить информацию о состоянии клиента. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса и идентификации клиента. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения;

- 3) Кэширование. Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некаэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные. Правильное использование кэширования помогает полностью или частично устранить некоторые клиент-серверные взаимодействия, ещё больше повышая производительность и расширяемость системы;
- 4) Единообразие интерфейса. К фундаментальным требованиям REST архитектуры относится и унифицированный, единообразный интерфейс. Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом. Каждый ресурс в REST должен быть идентифицирован посредством стабильного идентификатора, который не меняется при изменении состояния ресурса. Идентификатором в REST является URI;
- 5) Слои. Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда — просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования;
- 6) Код по требованию (необязательное ограничение). Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счёт загрузки кода с сервера в виде апплетов или сценариев.

В общем случае REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат. Как происходит управление информацией сервиса — это целиком и полностью основывается на протоколе передачи данных. Для HTTP

действие над данными задается с помощью методов: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить).

2.2 Формат передачи данных JSON

JSON (JavaScript Object Notation) – это текстовый формат представления данных в нотации объекта JavaScript. Предназначен JSON, также как и некоторые другие форматы такие как XML и YAML, для обмена данными. Несмотря на происхождение от JavaScript, формат считается независимым от языка и может использоваться практически с любым языком программирования. JSON основан на двух структурах данных:

- 1) Коллекция пар ключ/значение;
- 2) Упорядоченный список значений.

Это универсальные структуры данных. Почти все современные языки программирования поддерживают их в какой-либо форме. Объект JSON представляет собой заключённый в фигурные скобки список из пар ключ/значение. В коллекции ключ отделяется от значения с помощью знака двоеточия (:), а одна пара от другой - с помощью запятой (,). При этом ключ в JSON обязательно должен быть заключен в двойные кавычки. Значение ключа в JSON можно задать только в одном из следующих форматов: string (строкой), number (числом), object (объектом), array (массивом), boolean (логическим значением true или false) или null. Массив JSON представляет собой заключённый в квадратные скобки список из нуля или более значений, разделённых запятыми. Эти структуры могут быть вложенными.

2.3 Стандарт JSON Web Token

JSON Web Token (JWT) — это открытый стандарт (RFC 7519) представления данных для передачи между двумя или более сторонами в виде JSON-объекта. В клиент-серверных приложениях токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в

дальнейшем использует данный токен для подтверждения своей личности. Как правило, структурно JWT состоит из трех частей:

- 1) header — заголовок;
- 2) payload — полезная нагрузка;
- 3) signature — подпись.

Заголовок и полезная нагрузка — обычные JSON-объекты, которые необходимо дополнительно закодировать при помощи алгоритма base64url. Закодированные части соединяются друг с другом, и на их основе вычисляется подпись, которая также становится частью токена. Заголовок — служебная часть токена. Он помогает приложению определить, каким образом следует обрабатывать полученный токен:

```
{  
  "typ": "JWT",  
  "alg": "HS256"  
}
```

Поле `typ` определяет тип токена. Поле `alg` определяет алгоритм, использованный для генерации подписи. В приведенном случае был применен алгоритм HS256, в котором для генерации и проверки подписи используется единый секретный ключ. В полезной нагрузке передается любая информация, которая помогает приложению тем или иным образом идентифицировать пользователя и время жизни токена. Заголовок и полезная нагрузка кодируются при помощи алгоритма base64url, после чего объединяются в единую строку с использованием точки (".") в качестве разделителя. Генерируется подпись, которая добавляется к исходной строке так же через точку:

header.payload.signature

Получив JWT от пользователя, приложение самостоятельно вычислит значение подписи и сравнит его с тем значением, которое было передано в токене. Если эти значения не совпадут, значит, токен был модифицирован или сгенерирован недоверенной стороной, и принимать такой токен и доверять ему приложение не будет.

2.4 Шаблон проектирования MVVM

Паттерн MVVM позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения. MVVM паттерн был представлен Джоном Госсманом в 2005 году как модификация шаблона Presentation Model и был первоначально нацелен на разработку приложений в WPF[2]. Сейчас данный паттерн вышел за пределы WPF и применяется в самых различных технологиях, в том числе при разработке под Android. MVVM состоит из трех компонентов:

- 1) модели (Model);
- 2) модели представления (ViewModel);
- 3) представления (View)

На рисунке 1 представлен взаимодействие этих компонентов.

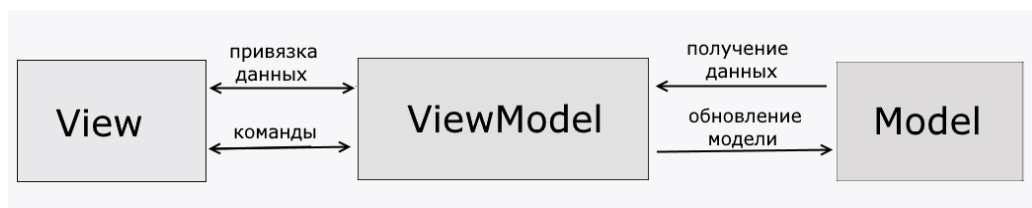


Рисунок 1 – Взаимодействие компонентов архитектуры MVVM

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную с этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

View или представление определяет визуальный интерфейс, через который пользователь взаимодействует с приложением.

ViewModel связывает модель и представление через механизм привязки данных. ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также ViewModel определяет логику по обновлению данных в модели.

2.5 Операционная система android

Android - операционная система для мобильных телефонов, смартфонов планшетов и других устройств, которая основывается на ядре Linux. Изначально ОС Android разрабатывала компания Android Inc., но затем ее купила компания Google. На рисунке 2 представлена архитектура операционной системы android.

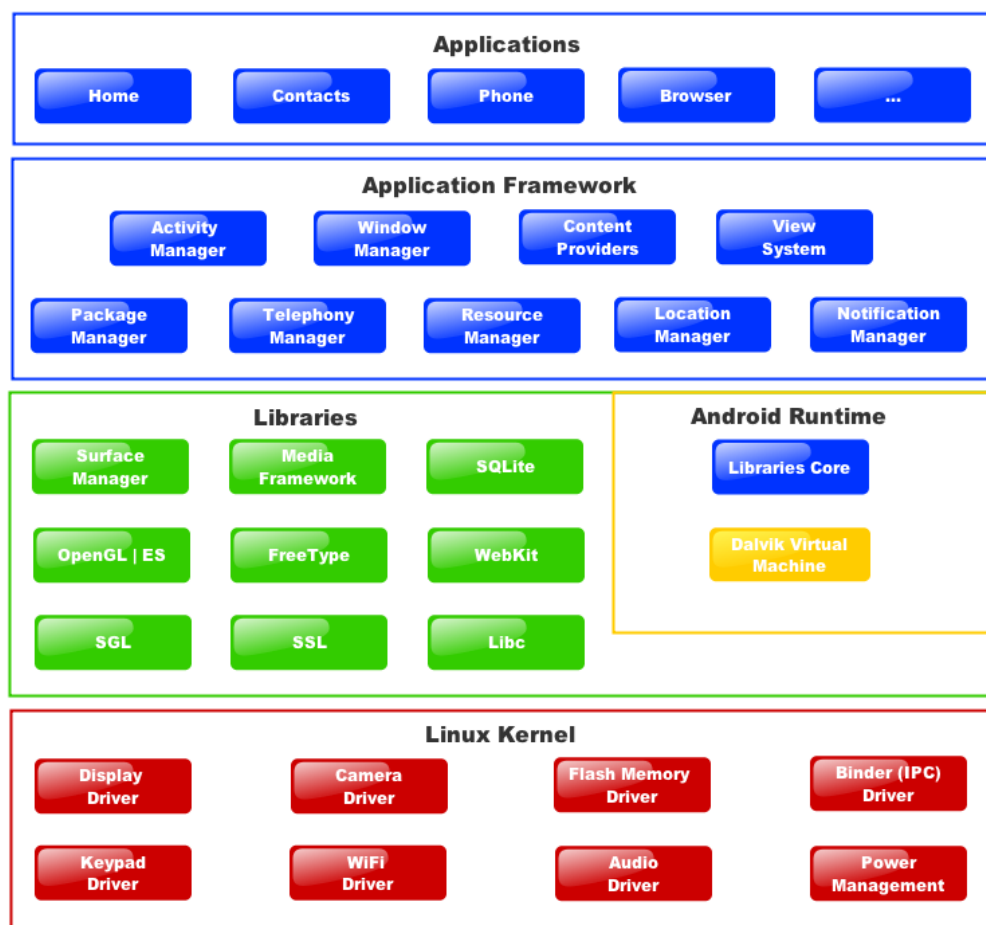


Рисунок 2 – Архитектура ОС Андроид

Уровень приложений (Applications). В состав Android входит комплект базовых приложений: клиенты электронной почты и SMS, календарь, различные карты, браузер, программа для управления контактами и много другое.

Уровень каркаса приложений (Application Framework). Android позволяет использовать всю мощь API, используемого в приложениях ядра. Архитектура построена таким образом, что любое приложение может использовать уже

реализованные возможности другого приложения при условии, что последнее откроет доступ на использование своей функциональности. Таким образом, архитектура реализует принцип многократного использования компонентов ОС и приложений. Основой всех приложений является набор систем и служб:

- 1) Система представлений (View System) – это богатый набор представлений с расширяемой функциональностью, который служит для построения внешнего вида приложений, включающий такие компоненты, как списки, таблицы, поля ввода, кнопки и т.п.
- 2) Контент-провайдеры (Content Providers) – это службы, которые позволяют приложениям получать доступ к данным других приложений, а также предоставлять доступ к своим данным.
- 3) Менеджер ресурсов (Resource Manager) предназначен для доступа к строковым, графическим и другим типам ресурсов.
- 4) Менеджер извещений (Notification Manager) позволяет любому приложению отображать пользовательские уведомления в строке статуса.
- 5) Менеджер действий (Activity Manager) управляет жизненным циклом приложений и предоставляет систему навигации по истории работы с действиями.

Уровень библиотек (Libraries). Платформа Android включает набор C/C++ библиотек, используемых различными компонентами ОС. Для разработчиков доступ к функциям этих библиотек реализован через использование Application Framework.

Уровень среды исполнения (Android Runtime). В состав Android входит набор библиотек ядра, которые предоставляют большую часть функциональности библиотек ядра языка Java. В Android используется собственная реализация JVM под названием Android Runtime (ART), специально оптимизированная для работы на мобильных устройствах. В старых версиях Android (до 5.0 Lollipop) вместо ART использовалась другая реализация под названием Dalvik.

Уровень ядра Linux (Linux Kernel). Android основан на ОС Linux, тем самым платформе доступны системные службы ядра, такие как управление памятью и процессами, обеспечение безопасности, работа с сетью и драйверами. Также ядро служит слоем абстракции между аппаратным и программным обеспечением.

2.6 Выбор и обоснование инструментов разработки

Исходя из системных требований предъявляемых к реализуемому серверу, к выбору инструментов разработки накладываются следующие ограничения:

- 1) Поддержка формата передачи данных JSON;
- 2) Наличие библиотек для работы со стандартом JWT;
- 3) Возможность реализации архитектурного стиля REST.

В качестве базы данных было решено использовать NoSQL СУБД MongoDB[3]. Подробное изучение данной СУБД позволило выявить ряд нижеописанных преимуществ для нашей системы:

- 1) Документоориентированность – СУБД предназначена для хранения иерархических структур данных (документов).
- 2) Представление в формате JSON – формат данных JSON удобен в интерпретации и использовании. В виду того, что серверная составляющая должна предоставлять RestAPI, MongoDB отлично подходит для реализации нашего сервиса.
- 3) Гибкость и динамичность данных – MongoDB способен хранить как структурированные, так и не структурированные данные.
- 4) Документация и ПО – данная СУБД является ПО с открытым исходным кодом, с отличной документацией и реализацией драйверов для подавляющего большинства языков программирования.

Исходя из системных требований к проектируемому сервису, представление данных в формате JSON стал решающим фактором в выборе

данного СУБД. Это позволит исключить использование дополнительных прослоек в виде ORM для преобразования данных.

Для реализации сервера используется язык программирования Python версии 3.8.5 и фреймворк для создания веб-приложений Flask[4]. Flask предоставляет гибкую настройку веб-приложения с минимальным количеством зависимостей. Пакетный менеджер pip предоставляет множество расширений для языка Python, благодаря чему реализация взаимодействия между компонентами системы сводится к написанию бизнес логики. Процесс написания программного кода и его отладки осуществлялся в интегрированной среде разработки PyCharm версии 2020.3.3. Для взаимодействия с базой данных используется рекомендованный способ для работы с MongoDB из среды Python пакет PyMongo[5]. Пример подключения к базе данных:

```
app.config["MONGO_URI"] = "mongodb://localhost:27017/barcode"
mongo = PyMongo(app)
```

Создается экземпляр класса клиента базы данных, который используется для обращения к данным. Благодаря тому, что данная база является документоориентированной с представлением объектов в типе BSON отсутствует миграция базы данных.

Для реализации стандарта аутентификации используется расширение Flask-JWT-Extended[6], которое реализует весь необходимый функционал. Для встраивания данного функционала в программный код, достаточно добавить к методам ресурсов соответствующие декораторы доступа. При возникновении ошибки декоратор прерывает выполнение обработки запроса и возвращает соответствующее сообщение. По умолчанию access-токен валиден в течение 15 минут, а refresh-токен в течение 30 дней. Для авторизованного запроса необходимо передать параметр Authorization в заголовке запроса с содержанием: Bearer <соответствующий токен доступа (access/refresh)>.

ГЛАВА 3. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ

Для проектирования системы использовался универсальный язык моделирования UML с использованием редактора диаграмм Visual Paradigm Community Edition v.16.2[7]. При проектировании системы, соответствующая вышеописанным требованиям, были разработаны следующие диаграммы:

- 1) диаграмма вариантов использования платформы;
- 2) диаграммы описания ресурсов RestAPI

3.1 Проектирование базы данных

В качестве СУБД для нашего проекта мы выбрали MongoDB. Актуальность выбора мы описали в предыдущей главе. Данное СУБД имеет следующие уровни представления данных:

- 1) Документ - JSON-объект имеющий произвольное число полей. Поля могут хранить как простое значение, так и вложенные объекты и массивы. Для каждого документа в MongoDB определен уникальный идентификатор. При добавлении документа в коллекцию данный идентификатор создается автоматически.
- 2) Коллекция (аналог таблиц в реляционных базах данных) - однотипные документы хранятся в отдельной коллекции. Документы в коллекции могут быть проиндексированы. Доступ к документу возможен как по ключу, так и по значению полей.
- 3) База данных - набор коллекций.

При проектировании нашей базы данных были определены следующие сущности требующие описания набора атрибутов:

- 1) Продукты
- 2) Пользователи
- 3) Токены

Контейнерами сущностей в MongoDB являются коллекции. Ниже представлены описание структуры документов для каждой коллекции с перечислением ключевых свойств.

Продукты

```
product = {
  "name": string,      - название продукта
  "barcode": string,   - штрих-код продукта
  "composition": [array], - состав продукта
  "comment": string,   - комментарии к продукту
  "gost": string,      - номер ГОСТ-стандарта
  "net mass": string,  - масса-нетто продукта
  "keeping time": string, - срок годности
  "storage conditions": string, - условия хранения
  "esl": {              - пищевая ценность продукта
    "protein": number, - количество белков на 100 грамм продукта
    "fats": number,   - количество жиров на 100 грамм продукта
    "carbohydrates": number, - количество углеводов на 100 грамм
    "calorie": number, - количество калорий на 100 грамм продукта
  }
  "packing type": string - тип упаковки
}
```

Пользователи

```
users = {
  "_id": Objectid, - уникальный идентификатор пользователя
  "name": string,   - имя пользователя (логин)
  "password": string, - хеш пароля пользователя
  "unacceptable_products": [array] - список недопустимых продуктов
}
```

Черный список токенов обновления

```
token_blacklist = {
  "_id": Objectid, - уникальный идентификатор
  "jti": string,   - токен обновления
  "created_at": date, - время добавления токена в формате UTC
  "user_id": Objectid - уникальный идентификатор пользователя
}
```

В черном списке будут храниться токены обновления пользователей с истекшим сроком жизни, чтобы исключить возможность повторного использования для получения токена доступа. По истечении срока жизни токены автоматически удаляются из коллекции.

3.2 Реализация серверной части проекта

Для описания общего представления о функциональном назначении платформы была разработана диаграмма вариантов использования представленная на рисунке 3. С платформой взаимодействует два актера:

- 1) гость – пользователь не авторизованный в системе;
- 2) авторизованный пользователь.

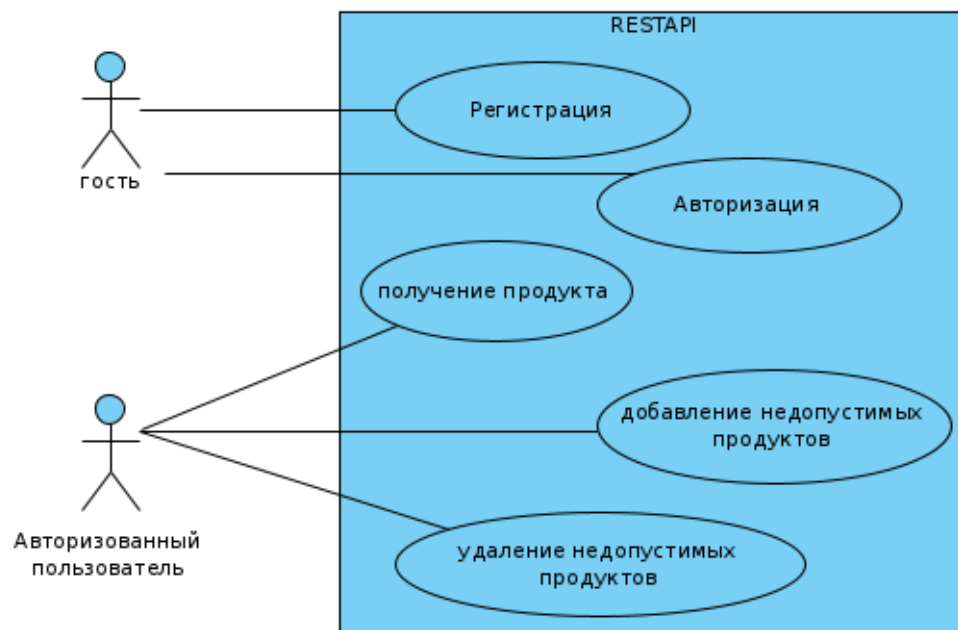


Рисунок 3 – Архитектура полной виртуализации

Гость платформы имеет следующие варианты использования:

- 1) регистрация в системе;
- 2) авторизация в системе.

Авторизованный пользователь имеет следующие варианты использования:

- 1) получение продукта – получение полной информации о продукте по штрих-коду и списка недопустимых продуктов присутствующих в составе;
- 2) добавление недопустимых продуктов – добавление нежелательного продукта в коллекцию;
- 3) удаление недопустимых продуктов – удаление нежелательного продукта из коллекции.

Для дальнейшего проектирования необходимо ввести понятие «ресурс». Ресурс в REST-архитектуре является ключевой абстракцией информации. Любая информация, которая может быть названа, может быть ресурсом. Сам Рой Филдинг в своей диссертации дает такое определение: «Любая информация, которая может быть названа, может быть ресурсом». Другими словами, любое понятие, которое может быть объектом гипертекстовой ссылки, должно вписываться в определение ресурса. Исходя из требований к реализуемому проекту можно выделить следующие ресурсы:

- 1) пользователи;
- 2) продукты;
- 3) недопустимые продукты;
- 4) токены.

Проектирование будем производить в среде visual paradigm, который предоставляет сервис для моделирования RestAPI. На рисунке 4 представлен модель RestAPI для ресурса пользователь. Для данного ресурса доступны два действия: регистрация и авторизация. На модели белая стрелка в черном квадрате направленная внутрь блока действия, обозначает запрос(request) к серверу, а обратная ответ(response). В заголовке блока действия указывается тип HTTP-запроса и URL. В синих блоках указываются параметры запроса и формат передаваемых параметров. При регистрации пользователя на сервер отправляется POST-запрос с необходимыми параметрами по указанному URL. В зависимости успешности действия возвращается соответствующий ответ. На рисунках 5 – 7 представлены модели для остальных ресурсов. Не

будем приводить описание для каждого ресурса, так как они аналогичны приведенному выше.

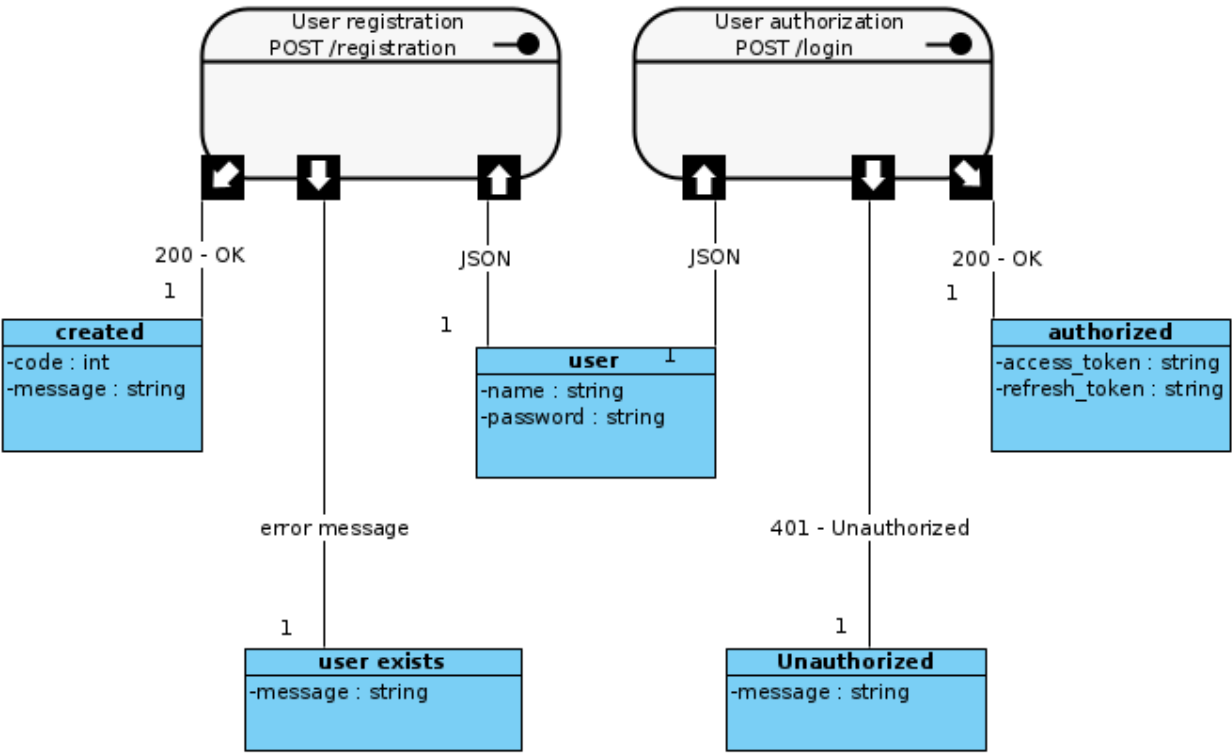


Рисунок 4 – Модель RestAPI для ресурса Пользователи

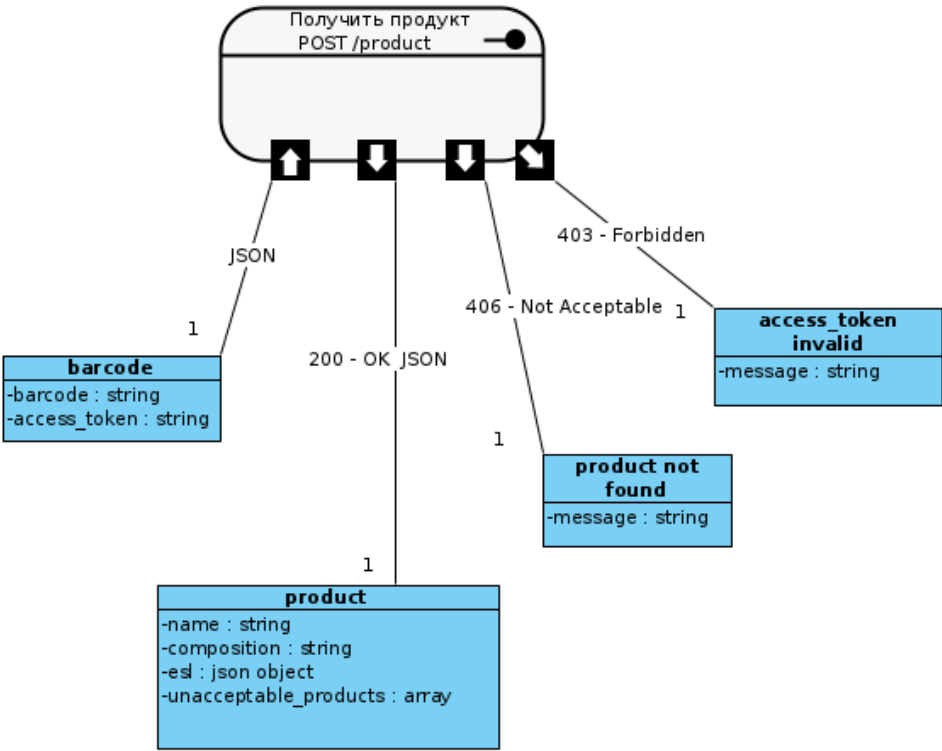


Рисунок 5 – Модель RestAPI для ресурса Продукты

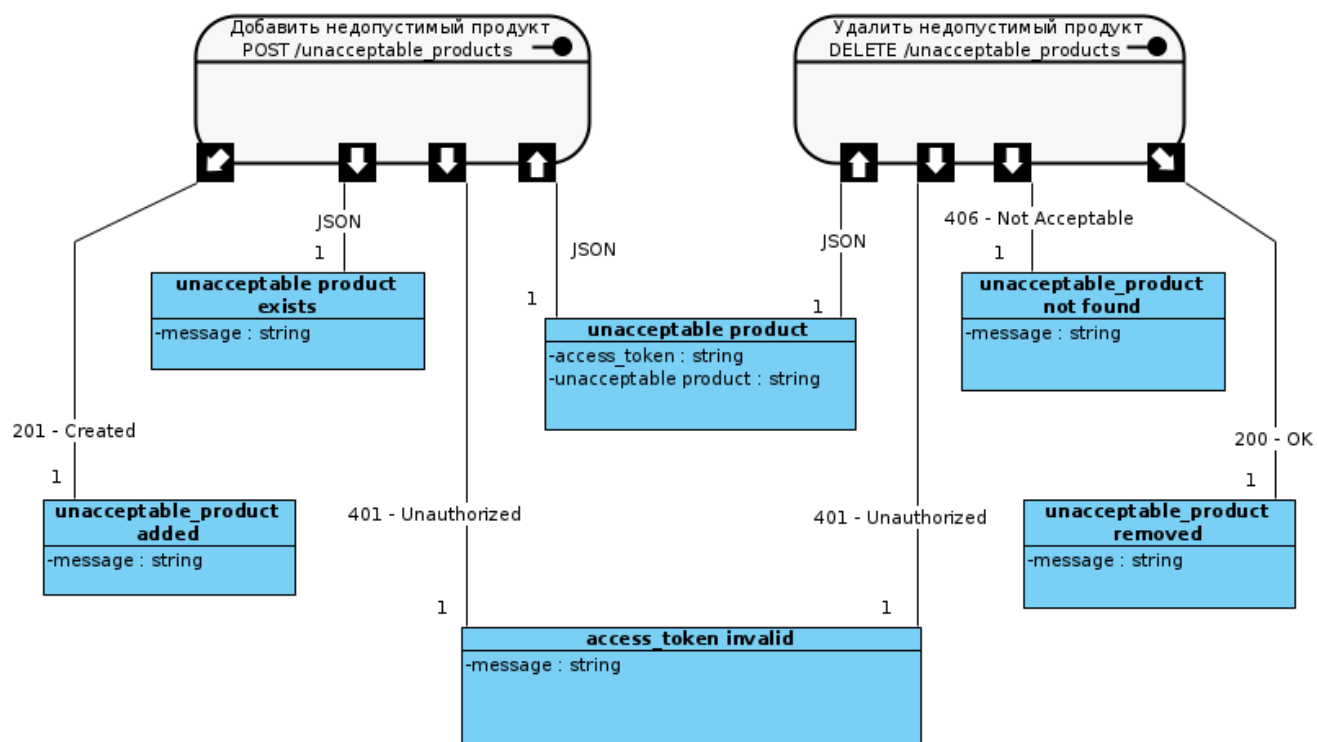


Рисунок 6 – Модель RestAPI для ресурса Недопустимые продукты

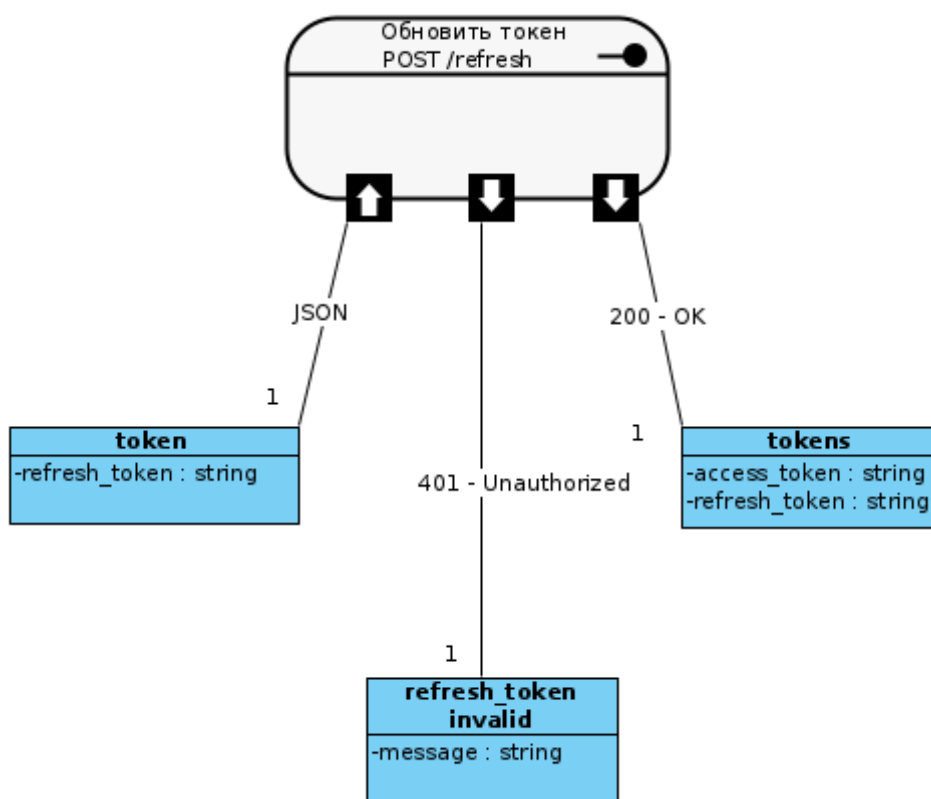


Рисунок 7 – Модель RestAPI для ресурса Токены

Опираясь на спроектированные модели, были разработаны методы для каждого ресурса требуемые для полноценной работы сервиса, которым присвоены соответствующие маршруты. В таблице 1 приводится описание всех реализованных методов. Листинг кода реализованного сервера приведен в приложении А. Ниже приводится описание основных параметров конфигурации реализованного сервера:

- 1) SECRET KEY – секретный ключ, который использует Flask при вычислении хэша паролей. Пакетное расширение jwt extended использует его для генерации подписи при создании токенов. Секретный ключ хранится отдельно от программного кода и передается посредством переменных окружения (environment variables), при развертке данного приложения на сервере;
- 2) JWT ACCESS TOKEN EXPIRES – срок жизни для токена доступа. Задаем равной 30 минутам;
- 3) JWT REFRESH TOKEN EXPIRES – срок жизни для токена обновления. Задаем равной 24 часам;
- 4) MONGO URI – URL сервера с MongoDB с необходимыми данными авторизации;

Все ресурсы на стороне сервера требуют авторизации с использованием токенов доступа, за исключением ресурсов, необходимых для аутентификации. Refresh-токен позволяет клиентам запрашивать новые access-токены по истечении их времени жизни. Refresh-токен выдается на более длительный срок, чем access-токен, и по истечению времени его жизни клиенту необходимо вновь пройти процесс аутентификации.

Таблица 1 – Описание реализованного RestAPI

URL	Метод	Описание
/registration	POST	Регистрация пользователя с переданными аргументами. Результатом является сообщение об успешной регистрации пользователя, либо сообщение о соответствующей ошибке с HTTP-кодом.
/login	POST	Аутентификация пользователя по логину и паролю. Результатом является access- и refresh-токены, либо сообщение о соответствующей ошибке с HTTP-кодом.
/refresh	POST	Обновление access-токена на основе переданного refresh-токена. Результатом является access-токен.
/unacceptable products	POST	Добавление недопустимого продукта для пользователя. Необходим заголовок авторизации. Результатом является сообщение об успешном добавлении продукта, либо сообщение о соответствующей ошибке с HTTP-кодом.
/unacceptable products	DELETE	Удаление недопустимого продукта из списка хранимой в базе данных. Необходим заголовок авторизации. Результатом является сообщение об успешном удалении продукта, либо сообщение о соответствующей ошибке с HTTP-кодом.
/product	POST	Получение информации о продукте по переданному штрих-коду. Необходим заголовок авторизации. Результатом является полная информация о продукте, либо сообщение о соответствующей ошибке с HTTP-кодом.

Источник: собственная разработка

В этом параграфе была спроектирована и реализована серверная часть приложения соответствующая архитектурному стилю REST. Отображена диаграмма вариантов использования сервиса, разработаны модели для каждого ресурса и реализованы соответствующие для них методы.

3.3 Реализация клиентской части проекта

Согласно требованиям архитектура приложения должна соответствовать шаблону проектирования MVVM[8]. На рисунке 8 представлена структура нашего приложения. Для изображения диаграммы использована среда проектирования Visual Paradigm Community Edition v.16.2.

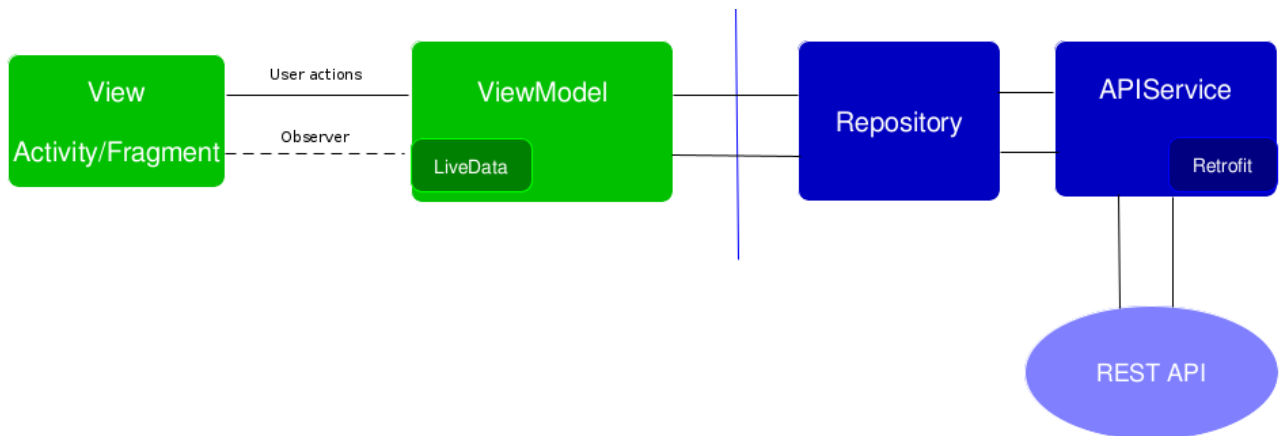


Рисунок 8 – Архитектура приложения

В классе ViewModel будет реализована логика приложения. View отвечает за работу с компонентами пользовательского интерфейса. Для отображения в пользовательском интерфейсе данных из ViewModel применяется хранилище данных LiveData[9]. LiveData - хранилище данных, работающее по принципу паттерна Observer (наблюдатель). Это хранилище умеет делать две вещи:

- 1) В него можно поместить какой-либо объект;
- 2) На него можно подписаться и получать объекты, которые в него помещают.

Activity подписывается на LiveData и получает данные, которые помещает в него ViewModel. В Repository будут реализованы методы для получения данных из нашего сервера. Для описания и отправки HTTP-запросов применяется библиотека Retrofit[10]. Интерфейсы всех необходимых запросов описываются в APIService.

Пользовательский интерфейс приложения состоит из 5 страниц(Fragments) наследуемые от одного Activity:

- 1) Главная;
- 2) Авторизация;
- 3) Регистрация;
- 4) Состав продукта;
- 5) Список недопустимых продуктов.

На рисунке 9 представлен разработанный навигационный граф нашего приложения.

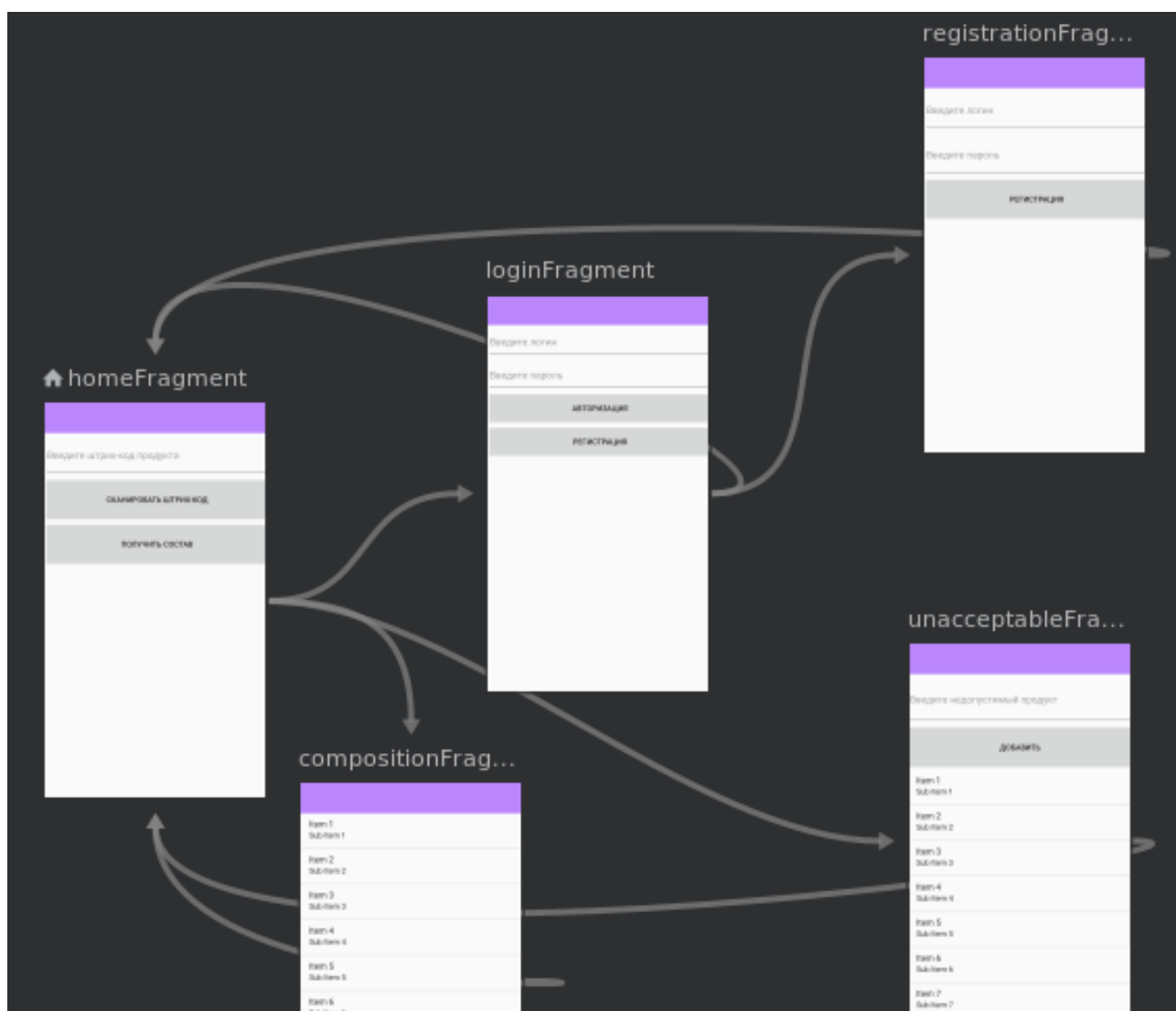


Рисунок 9 – Навигационный граф приложения

Рассмотрим подробнее реализованные страницы приложения. На рисунке 10 представлена разработанная домашняя страница клиентского приложения. Данная страница содержит следующие компоненты:

- 1) поле для ввода штрих-кода продукта;
- 2) кнопку для сканирования штрих-кода через камеру на телефоне
- 3) кнопку получить состав

При активации кнопки «Получить состав» отправляется HTTP-запрос на сервер для получения необходимой информации о продукте и переход на страницу «Состав продукта». Для получения успешного ответа от сервера в заголовке запроса необходимо передать токен доступа, для этого пользователь должен быть авторизован. Страница авторизации представлена на рисунке 12.

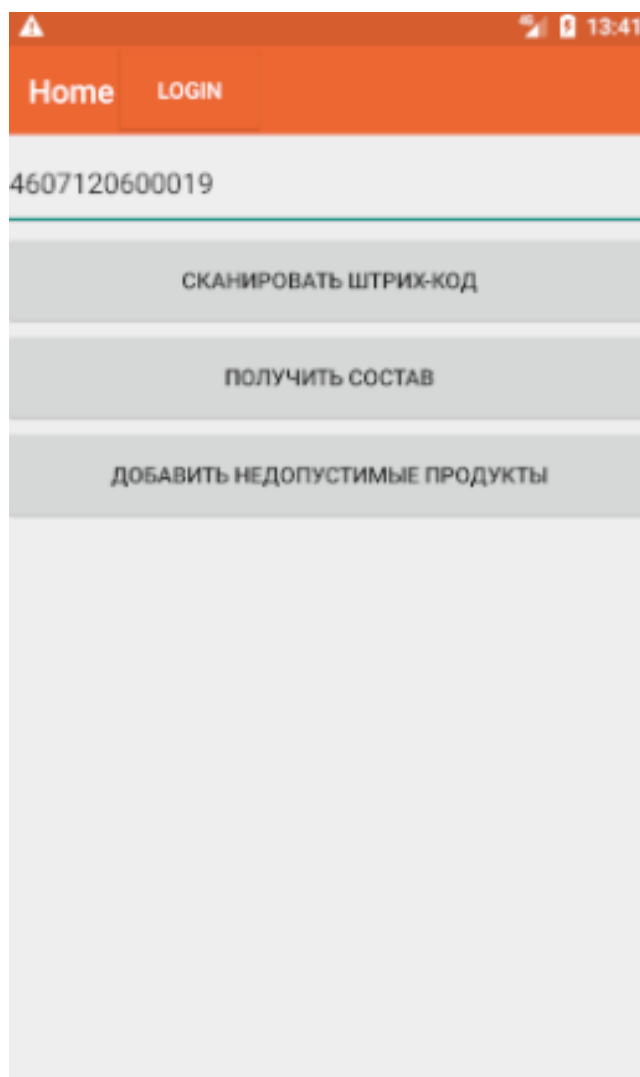


Рисунок 10 – Домашняя страница

При условии отсутствия исключения вовремя выполнения запроса, ViewModel получает информацию о продукте от сервера и заносит в хранилище LiveData. Наблюдатель, «Состав продукта» представленный на рисунке 11, видит изменения в хранилище данных и отображает данные для пользователя.

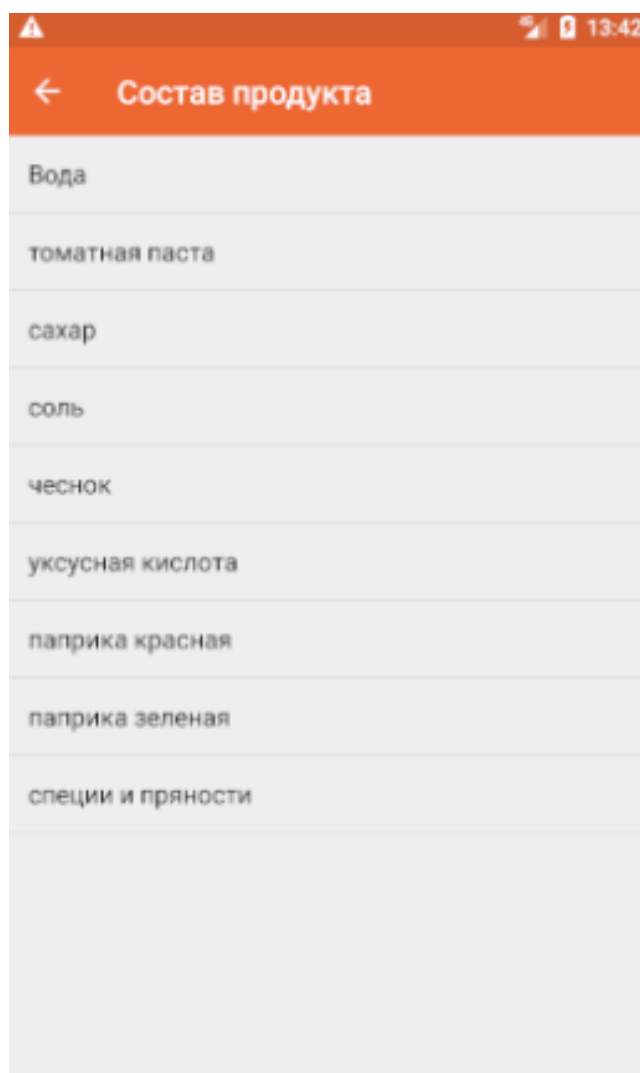
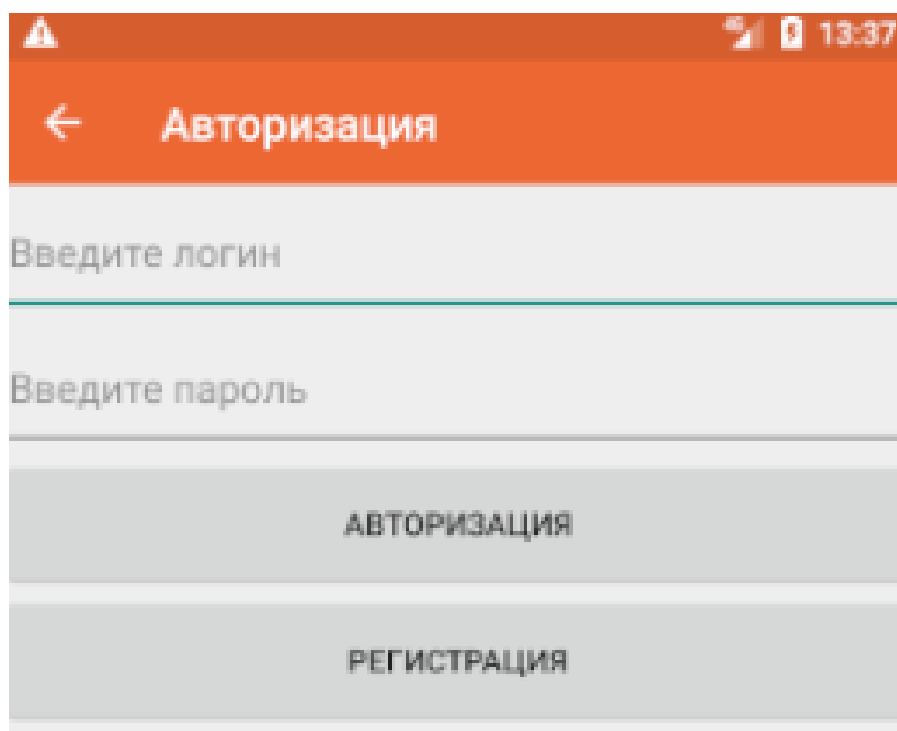


Рисунок 11 – Состав продукта

При авторизации пользователя ViewModel получает токены доступа и обновления. Для сохранения токенов и передачи в заголовок каждого запроса используется SharedPreferences[11]. SharedPreferences позволяет сохранять данные в виде пар ключ-значение. Android хранит SharedPreferences в виде XML файла в папке sharedprefs. Это позволяет исключить потерю токенов при закрытии приложения и отказаться от использования базы данных на стороне клиента. Использование приватного режима сохранения (PRIVATE), не позволит другим приложениям иметь доступ к этим файлам и обеспечит необходимую безопасность. Для регистрации пользователя на сервере необходимо активировать кнопку «Регистрация» и приложение перейдет на необходимую страницу. Пользовательский интерфейс страницы регистрации приведен на рисунке 13.



Авторизация

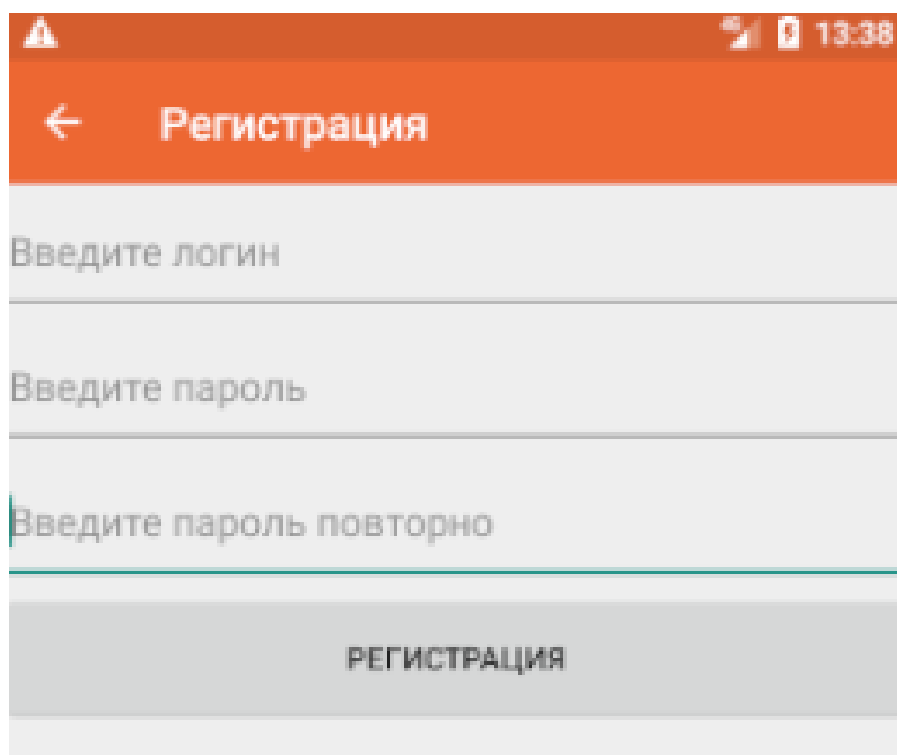
Введите логин

Введите пароль

АВТОРИЗАЦИЯ

РЕГИСТРАЦИЯ

Рисунок 12 – Страница авторизации



Регистрация

Введите логин

Введите пароль

Введите пароль повторно

РЕГИСТРАЦИЯ

Рисунок 13 – Страница регистрации

На рисунке 14 представлена страница добавления недопустимых для пользователя продуктов.

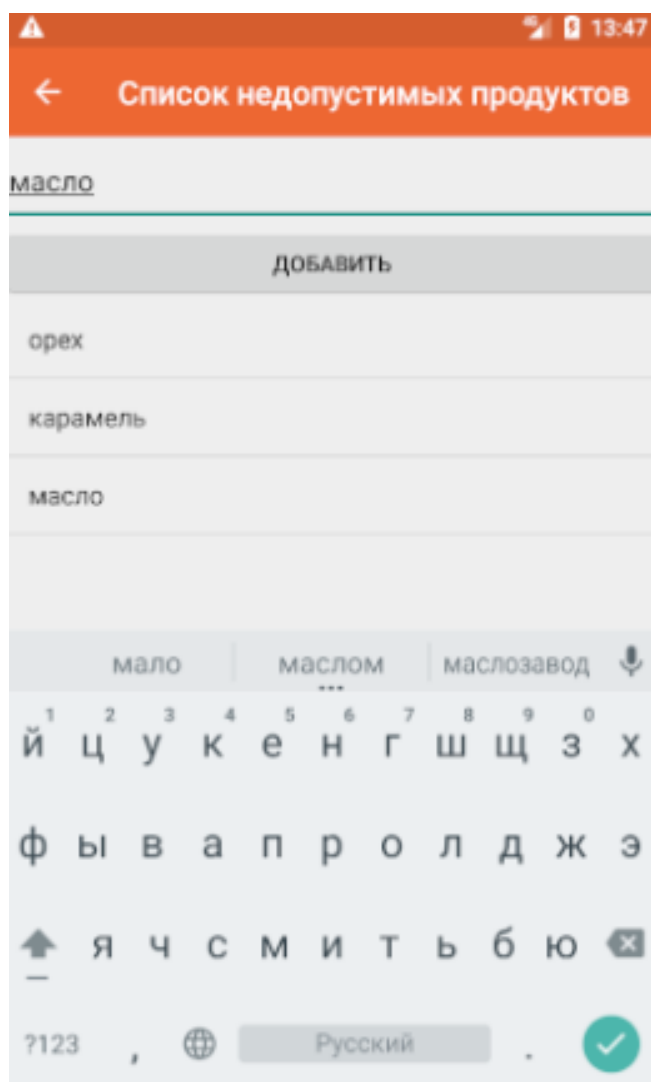


Рисунок 14 – Добавление недопустимых продуктов

Для добавления продуктов в список недопустимых, необходимо ввести в поле ввода нежелательный продукт и активировать кнопку «Добавить». Сервер обработает полученный запрос от клиента, обновит в базе данных список и отправит для отображения пользователю.

Основные и наиболее важные фрагменты реализованного кода клиентской части приведены в приложении Б.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы было разработано приложение для ускорения процесса выбора и повышения качества потребляемых продуктов в любых продовольственных магазинах.

В основу разработанной системы была положена технология клиент-серверной архитектуры. При реализации серверной части придерживались шаблона проектирования REST. Архитектура клиентской части соответствует шаблону проектирования MVVM.

В ходе выполнения работы были решены следующие задачи:

- 1) Определены требования к сервису;
- 2) Выполнен обзор требуемых технологий;
- 3) Выполнено проектирование;
- 4) Разработана серверная часть приложения полностью соответствующая предъявленным требованиям.
- 5) Реализован рабочий прототип клиентского приложения.

Реализованная версия клиентской части приложения не является финальной и подлежит дальнейшей доработке.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Roy Thomas Fielding Architectural Styles and the Design of Network-based Software Architectures [Электронный ресурс] Режим доступа: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- 2 Шаблон Model-View-ViewModel - Xamarin | Microsoft Docs [Электронный ресурс] Режим доступа: <https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/enterprise-application-patterns/mvvm>
- 3 MongoDB Documentation [Электронный ресурс] Режим доступа: <https://docs.mongodb.com/>
- 4 Flask Documentation [Электронный ресурс] Режим доступа: <https://flask.palletsprojects.com/en/1.1.x/>
- 5 Flask-PyMongo 2.3.0 documentation [Электронный ресурс] Режим доступа: <https://flask-pymongo.readthedocs.io/en/latest/>
- 6 Flask-JWT-Extended's Documentation [Электронный ресурс] Режим доступа: <https://flask-jwt-extended.readthedocs.io/en/stable/>
- 7 Visual Paradigm User's Guides [Электронный ресурс] Режим доступа: <https://www.visual-paradigm.com/support/documents/>
- 8 Guide to app architecture | Android Developers [Электронный ресурс] Режим доступа: <https://developer.android.com/jetpack/guide>
- 9 LiveData Overview | Android Developers [Электронный ресурс] Режим доступа: <https://developer.android.com/topic/libraries/architecture/livedata>
- 10 Retrofit - Square Open Source [Электронный ресурс] Режим доступа: <https://square.github.io/retrofit/>
- 11 SharedPreferences | Android Developers [Электронный ресурс] Режим доступа: <https://developer.android.com/reference/android/content/SharedPreferences>
- 12 Android Developers Docs [Электронный ресурс] Режим доступа: <https://www.google.com/search?q=ANDROID+DEVELOPER>

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Листинг кода серверной части приложения

```
import datetime
import os
import re

from bson.objectid import ObjectId
from flask import Flask, jsonify, request, make_response
from flask_jwt_extended import JWTManager
from flask_jwt_extended import create_access_token,
    create_refresh_token, get_jwt
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_pymongo import PyMongo
from werkzeug.security import generate_password_hash,
    check_password_hash

app = Flask(__name__)
app.config['SECRET_KEY'] = os.getenv('SECRET_KEY', '
    DefaultValue_d5049c7d')
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = datetime.timedelta(minutes
    =30)
app.config["JWT_REFRESH_TOKEN_EXPIRES"] = datetime.timedelta(hours
    =24)
app.config["MONGO_URI"] = "mongodb://localhost:27017/barcode"
app.config['DEBUG'] = True

mongo = PyMongo(app)
jwt = JWTManager(app)
current_time = datetime.datetime.utcnow()
```

Продолжение Приложения А

```
# Регистрация пользователя
@app.route('/registration', methods=['POST'])
def create_user():
    data = request.get_json()
    existing_user = mongo.db.users.find_one({'name': data['name']})
    if existing_user is None:
        hash_pass = generate_password_hash(data['password'], method='sha256
            ')
        user = {
            'name': data['name'],
            'password': hash_pass,
            "unacceptable_products": [],
        }
        mongo.db.users.insert(user)
        user = mongo.db.users.find_one({'name': data['name']}, {'_id': 1, '
            name': 1, 'password': 1})
        access_token = create_access_token(identity=str(user['_id']))
        refresh_token = create_refresh_token(identity=str(user['_id']))
        return jsonify(access_token=access_token, refresh_token=
            refresh_token)
    return 'User exist'

# Авторизация пользователя
@app.route('/login', methods=['POST'])
def user_login():
    data = request.get_json()
    if not data or not data['name'] or not data['password']:
        return make_response('Could not verify', 401)
    user = mongo.db.users.find_one({'name': data['name']}, {'_id': 1, '
        name': 1, 'password': 1})
    if not user:
        return make_response('Could not verify', 401)
    if check_password_hash(user['password'], data['password']):
        access_token = create_access_token(identity=str(user['_id']))
        refresh_token = create_refresh_token(identity=str(user['_id']))
```

Продолжение Приложения А

```
return jsonify(access_token=access_token, refresh_token=
    refresh_token)
return make_response('Could not verify', 401)

# Обновление токена доступа
@app.route("/refresh", methods=["POST"])
@jwt_required(refresh=True)
def refresh():
    current_user = get_jwt_identity()
    jti = get_jwt()["jti"]
    existing_jti = mongo.db.token_blacklist.find_one({'user_id':
        current_user, 'jti': jti})
    if existing_jti:
        return jsonify('this token invalid')
    token_blacklist = {
        'jti': jti,
        'created_at': current_time,
        'user_id': current_user }
    mongo.db.token_blacklist.insert(token_blacklist)
    access_token = create_access_token(identity=current_user)
    refresh_token = create_refresh_token(identity=current_user)
    return jsonify(access_token=access_token, refresh_token=
        refresh_token)

# Добавление и удаление недопустимых продуктов
@app.route("/unacceptable_products", methods=['POST', 'DELETE'])
@jwt_required()
def add_unacceptable_products():
    current_user = get_jwt_identity()
    unacceptable_products = request.json['unacceptable_products']
    if request.method == 'POST':
        mongo.db.users.update({"_id": ObjectId(current_user)},
            {'$addToSet': {'unacceptable_products': {'$each':
                unacceptable_products}}})
```

Продолжение Приложения А

```
return jsonify("Указанные продукты добавлены в список  
    нежелательных")  
if request.method == 'DELETE':  
    mongo.db.users.update({"_id": ObjectId(current_user)},  
    {'$pullAll': {'unacceptable_products': unacceptable_products}})  
    return jsonify("Указанные продукты удалены из списка  
        нежелательных")  
  
# получение продукта по штрих-коду  
@app.route("/product", methods=['POST'])  
@jwt_required()  
def test_req():  
    current_user = get_jwt_identity()  
    barcode = request.json['barcode']  
    product = mongo.db.goods.find_one_or_404({"barcode": barcode},  
    {'composition': 1, '_id': 0, 'name': 1, 'esl': 1})  
    unacceptable_products = mongo.db.users.find_one_or_404({"_id":  
        ObjectId(current_user)},  
    {'unacceptable_products': 1})  
    # result = list(set(product['composition']) & set(  
        unacceptable_products['unacceptable_products']))  
    result = []  
    for unacceptable in unacceptable_products['unacceptable_products']:  
        for composition in product['composition']:  
            temp = re.search(f'(?i){unacceptable}', composition)  
            if temp is not None:  
                result.append(composition)  
    return_result = {  
        "name": product['name'],  
        "composition": product['composition'],  
        "unacceptable_products": result,  
        "esl": product['esl'] }  
    return jsonify(return_result)  
  
if __name__ == '__main__':  
    app.run()
```

ПРИЛОЖЕНИЕ Б

Основные фрагменты реализованного кода клиентской части приложения

```
# Интерфейсы для составления HTTP-запроса на сервер
ApiService.kt

interface Login {
    @POST("login")
    suspend fun getToken(
        @Body login: LoginRequest
    ): Response<LoginResponse>
}

interface Registration {
    @POST("registration")
    suspend fun getToken(
        @Body login: LoginRequest
    ): Response<LoginResponse>
}

interface UnacceptableProductPost {
    @POST("unacceptable_products")
    suspend fun setUnacceptableProduct(
        @Header("Authorization") token: String,
        @Body unacceptableProduct: UnacceptableProductRequest
    ): Response<UnacceptableProductResponse>
}

interface UnacceptableProductDelete {
    @DELETE("unacceptable_products")
    suspend fun delUnacceptableProduct(
        @Header("Authorization") token: String,
        @Body unacceptableProduct: UnacceptableProductRequest
    ): Response<UnacceptableProductResponse>
}
```


Продолжение Приложения Б

```
interface Product {
    @POST("product")
    suspend fun getProduct(
        @Header("Authorization") token: String,
        @Body barcode: Barcode
    ): Response<Product>
}

interface Refresh {
    @POST("refresh")
    suspend fun getRefreshToken(
        @Header("Authorization") refreshToken: String
    ): Response<LoginResponse>
}

# Методы для получения данных с сервера
class Repository {

    suspend fun getToken(login: LoginRequest): Response<LoginResponse>{
        return RetrofitBuilder.login.getToken(login)
    }

    suspend fun getTokenReg(login: LoginRequest): Response<
        LoginResponse>{
        return RetrofitBuilder.registration.getToken(login)
    }

    suspend fun setUnacceptableProduct(token: String,
        unacceptableProduct: UnacceptableProductRequest ): Response<
        UnacceptableProductResponse>{
        return RetrofitBuilder.unacceptableProductPost.
            setUnacceptableProduct(token, unacceptableProduct)
    }
}
```

Продолжение Приложения Б

```
suspend fun delUnacceptableProduct(token: String,
    unacceptableProduct: UnacceptableProductRequest ): Response<
    UnacceptableProductResponse>{
return RetrofitBuilder.unacceptableProductDelete.
    delUnacceptableProduct(token, unacceptableProduct)
}

suspend fun getProduct(token: String, barcode: Barcode): Response<
    Product>{
return RetrofitBuilder.product.getProduct(token, barcode)
}

suspend fun getRefreshToken(refreshToken: String): Response<
    LoginResponse>{
return RetrofitBuilder.refresh.getRefreshToken(refreshToken)
}
}

# Реализованные сопрограммы(корутины) для отправки асинхронных
  запросов
class MainViewModel(private val repository: Repository): ViewModel
    () {

var product: MutableLiveData<Response<Product>> = MutableLiveData()
var accessToken: MutableLiveData<String> = MutableLiveData()
var refreshToken: MutableLiveData<String> = MutableLiveData()
var unacceptable: MutableLiveData<Response<
    UnacceptableProductResponse>> = MutableLiveData()

fun getToken(login: LoginRequest){
viewModelScope.launch {
val responseTokens : Response<LoginResponse> = repository.getToken(
    login)
if (responseTokens.isSuccessful) {
```

Продолжение Приложения Б

```
val access_token: String = responseTokens.body()?.access_token.
    toString()
accessToken.value = access_token
val refresh_token: String = responseTokens.body()?.refresh_token.
    toString()
refreshToken.value = refresh_token
}
}
}

fun getTokenReg(login: LoginRequest){
viewModelScope.launch {
val responseTokens : Response<LoginResponse> = repository.
    getTokenReg(login)
if (responseTokens.isSuccessful) {
val access_token: String = responseTokens.body()?.access_token.
    toString()
accessToken.value = access_token
val refresh_token: String = responseTokens.body()?.refresh_token.
    toString()
refreshToken.value = refresh_token
}
}
}

fun setUnacceptableProduct(token: String, unacceptableProduct:
    UnacceptableProductRequest){
viewModelScope.launch {
val responseUnacceptableProduct : Response<
    UnacceptableProductResponse> = repository.setUnacceptableProduct
        (token, unacceptableProduct)
if (responseUnacceptableProduct.isSuccessful) {
unacceptable.value = responseUnacceptableProduct
}
}
}
```

Продолжение Приложения Б

```
fun delUnacceptableProduct(token: String, unacceptableProduct:
    UnacceptableProductRequest){
viewModelScope.launch {
val responseUnacceptableProduct : Response<
    UnacceptableProductResponse> = repository.delUnacceptableProduct
    (token, unacceptableProduct)
if (responseUnacceptableProduct.isSuccessful) {
unacceptable.value = responseUnacceptableProduct
}
}
}

fun getProduct(token: String, barcode: Barcode){
viewModelScope.launch {
val responseProduct : Response<Product> = repository.getProduct(
    token, barcode)
if (responseProduct.isSuccessful) {
product.value = responseProduct
}
}
}

fun getRefreshToken(refToken: String){
viewModelScope.launch {
val refreshTokens: Response<LoginResponse> = repository.
    getRefreshToken(refToken)
if (refreshTokens.isSuccessful) {
val access_token: String = refreshTokens.body()?.access_token.
    toString()
accessToken.value = access_token
val refresh_token: String = refreshTokens.body()?.refresh_token.
    toString()
refreshToken.value = refresh_token
}
}
}}
```