МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
«Гродненский государственный университет имени Янки Купалы»

Факультет математики и иформатики
Кафедра современных технологий программирования

ВАСИЛЬКОВ ВЛАДИМИР ЮРЬЕВИЧ

Разработка комплекса приложений ”Оптовая торговля”

Дипломная работа
студента 4 курса специальности
1-40 01 01 «Программное обеспечение информационных технологий»
дневной формы получения образования

«Допустить к защите»
Заведующий кафедрой
_____ Рудикова Л. В.
«___» _____ 2020 г.

Научный руководитель
Гуща Юлия Вальдемаровна,
старший преподаватель кафедры
современных технологий
программирования

Гродно 2020

РЕЗЮМЕ

Васильков Владимир Юрьевич

Разработка комплекса приложений ”Оптовая торговля”

72 страниц, 12 рисунков, 9 приложений, 5 листингов, 11 использованных источников литературы.

Ключевые слова: сервер, интернет-магазин, мобильное приложение, веб-приложение, язык программирования Kotlin, REST-архитектура, контроль доступа, клиент-серверная архитектура, Spring Boot, JSON, веб-клиент, язык программирования TypeScript, Angular, HTML, CSS, Android-клиент, Android, MVVM, MVC, Clean Architecture, Android Architecture Components, базы данных.

Цель дипломной работы – проектирование и разработка клиент-серверных приложений взаимодействующих посредством REST-принципа.

Задача дипломной работы – реализовать комплекс приложений, способных взаимодействовать между собой, позволяющий управлять данными, которые содержатся в базе данных, контролировать доступ и привилегии пользователей, состоящий из серверного приложениями и нескольких клиентских приложений использующих разные платформы.

Объектом исследования выступают REST-приложения, построенные по принципу клиент-сервер.

Предмет исследования – основные функции и принципы функционирования клиент-серверных приложений, использующих REST-подход для взаимодействия клиента и сервера.

SUMMARY

Uladzimir Vasilkou Yurievich

Development of set of applications ”Wholesale online store”

72 pages, 12 images, 9 extensions, 5 listings, 11 bibliography sources.

Keywords: server, online store, mobile application, web-application, Kotlin programming language, REST-architecture, access control, client-server architecture, Spring Boot, JSON, web-client, TypeScript programming language, Angular, HTML, CSS, Android-client, Android, MVVM, MVC, Clean Architecture, Android Architecture Components, databases.

The purpose of the graduate work is investigation and research methods of implementation client-server application systems that interact with the help of REST-principle.

The aim of the graduate work is to implement application set, that can interact with each other, control data, that are stored in database, control access and user’s privileges, that contains server application and few client applications that are uses different platforms.

The object of reserch is REST-application, that interact with the help of client-server principle.

The research subject is the main function and principles functioning client-server applications that uses REST way of interruction between client and server.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

API (Application Programming Interface) – набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой) для использования во внешних программных продуктах. Часто выполняет роль слоя абстракции, упрощающего доступ к функциям приложения (библиотеки). Используется для написания всевозможных приложений, основанных на готовом программном решении [1].

ПО (Программное обеспечение) – совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.

REST API (Representational State Transfer) — архитектурный стиль взаимодействия компонентов клиент-серверного приложения в сети. Такой подход помогает поддерживать несколько клиентских приложений на разных платформах, а также позволяет поддерживать достаточный уровень абстрагированности и масштабируемости. Представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой системы. В определённых случаях это приводит к повышению производительности и упрощению архитектуры.

UI (user interface, пользовательский интерфейс) – способ взаимодействия между пользователем и приложением.

CMS (content management system, система управления контентом) – приложение либо их связка, которые используются для создания и управления цифровым контентом.

JVM (Java Virtual Machine) – виртуальная машина которая позволяет компьютеру запускать Java приложения ровно так же, как и программы которые

были написаны на других языках, которые компилируются в Java байт-код [2].

DI (Dependency Injection) – подход использующийся в объектно-ориентированных языках, для внедрения необходимых зависимостей.

IoC (Inversion of Controle) – принцип использующийся в объектно-ориентированных языках, для повышения модульности приложения и возможности сделать его расширяемым. Суть заключается в разделении зависимостей между высокоуровневыми и низкоуровневыми слоями приложения через ряд абстракций.

SRP (Single Responsibility Principle) – принцип примененение которого подразумевает, что каждый модуль или класс должен иметь лишь выполняемую задачу, которая инкапсулированна в классе, модуле или функции [3].

JPA (Java Persistence API) – Java EE/SE спецификация, описывающая систему управления сохранением Java объектов в таблицы реляционных БД в удобном виде, с помощью аннотаций [4].

БД (база данных) – организованная и структурированная коллекция данных, которая обычно хранится и доступна через компьютерную систему.

DSL (Domain Specific Language, предметно-ориентированный язык) – язык программирования разработанный для решения определённого (крайне узкого) списка задач.

SMPT (Simple Mail Transfer Protocol, простой протокол передачи почты) – широко использующийся сетевой протокол, предназначеный для передачи почтовых e-mail сообщений в сетях TCP/IP [5].

HTTP (Hypertext Transfer Protocol, протокол пересылки гипертекста) – протокол для пересылки гипермедиа документов, таких как HTML. Был разработан для сообщения между веб-браузерами и веб-серверами, однако может быть использован и для других задач [6].

OAuth – открытый протокол авторизации, который позволяет предоставить третьей стороне ограниченный доступ к защищенным ресурсам пользователя без необходимости передавать ей логин и пароль.

ВВЕДЕНИЕ

В связи с развитием компьютерных технологий, жизнь человека упрощается и появляются новые решения, которые делают её лучше и удобнее. Раньше, для покупки необходимого товара, человеку требовалось покидать свой дом и отправляться на его поиски. В эпоху Интернет-технологий стали появляться интернет-магазины, которые предоставили возможность гораздо быстрее находить и получать желаемые товары и продукты, оставаясь дома.

В современном мире уже сложно найти достаточно крупную организацию, которая не имеет в своём распоряжении работающий интернет-магазин. Хорошо налаженный интернет магазин может повысить производительность организации в целом и сократить некоторые расходы.

Одним из возможных решений может являться создание отдельного сервера, который будет возвращать единые, для всех платформ данные. В таком случае, любое клиентское приложение будет работать с единой базой данных и т.д., а поведение приложения будет отличаться только в случае различной реализации клиентского приложения.

Для достижения поставленной цели предусмотрены решения следующих задач:

1. Анализ и формулировка требований к разрабатываемым приложениям.
2. Обзор и выбор средств разработки.
3. Проектирование архитектуры приложения.
4. Проектирование БД.
5. Проектирование API.
6. UI/UX дизайн.
7. Реализация приложения REST-сервера.
8. Реализация веб-клиента.
9. Реализация Android-клиента.

ГЛАВА 1
АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Основные сведения

Интернет-магазин – сайт, торгующий товарами посредством сети Интернет. Позволяет пользователям, сформировать заказ на покупку онлайн. Типичный интернет-магазин позволяет клиенту просматривать ассортимент продуктов и услуг фирмы, фотографии или изображения продуктов, а также информацию о технических характеристиках продуктов и ценах. Интернет-магазины обычно позволяет покупателям использовать функции поиска.

Клиент-сервер – вычислительная или сетевая архитектура, в которой задачи распределены между поставщиками услуг (сервера), и заказчиками (клиенты).

Клиент и сервер являются ПО, которое расположено на разных вычислительных машинах и взаимодействуют друг с другом с помощью вычислительной сети, посредством сетевых протоколов. Серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных, или в виде сервисных функций. Обычно, программу-сервер размещают на специально выделенном вычислительном устройстве, которое настроено особым образом, т.к. сервер может выполнять запросы от многих программ-клиентов и его производительность должна быть высокой [7].

К достоинствам клиент-серверной архитектуры относят:

- отсутствие дублирования кода программы-сервера программами-клиентами;
- снижение требований к клиентским устройствам, т.к. все вычисления выполняются на стороне сервера;

- все данные хранятся на сервере, который защищен гораздо лучше большей части клиентов;
- возможность организации контроля полномочий, чтобы предоставлять доступ клиентам с определёнными полномочиями.

К недостаткам относят:

- Поломка на стороне сервера, может привести к неработоспособности всей сети приложений;
- Поддержка работы системы требует отдельного специалиста – системного администратора;
- Высокая стоимость сетевого оборудования.

1.2 Обзор подходов реализации клиент-серверных приложений

Для построения подобных систем приложений могут использоваться множество разнообразных подходов с разным списком технологий. Так например, глобальная архитектура системы приложений может быть монолитной, модульной либо ориентированой на сервисы.

- Монолитный подход является наиболее старой моделью построения ПО. Именно с такой архитектурой начиналась разработка любого ПО. Используя данный подход можно избежать сложную архитектуру веб-приложения, поскольку веб-сервер будет содержать в себе весь функционал необходимый для реализации бизнес логики, а база данных будет предоставлять необходимые данные серверу. Однако данная архитектура не является хорошо масштабируемой и в следствии развития приложения, может накопиться множество технических долгов, которые в последствии будет значительно сложнее исправлять.
- Модульная архитектура, подразумевает под собой разделение функционала приложения на отдельные модули, каждый из которых ответственен за определённую часть функционала. Так, получается приложение, состоящее из большого колличества монолитных

модулей внутри одного приложения и каждый из модулей получается функционально независимым от других подобных модулей. Однако с таким подходом реализации архитектуры, стоит учитывать, что даже на начальном этапе разработки, стоимость на порядок выше, чем при применении монолитной архитектуры.

- Сервисы представляют собой отдельные, самодостаточные модули, обладающие своей аппаратной базой (в т.ч. каждый может обладать своей базой данных). Таким образом, взаимодействие между сервисам происходит асинхронно, что позволяет достичь независимого масштабирования компонентов приложения (опираясь на необходимости конкретного сервиса) и позволяет использовать различные языки программирования для реализации каждого из сервисов. Однако, такая архитектура является наиболее дорогой в разработке и помимо этого, обязывает строго определить и следовать правилам взаимодействия между сервисами, их API и модели данных.

Для реализации пользовательского интерфейса могут быть использованы как простая комбинация языков разметки, стилей и скриптов, так и специализированные фреймворки для построения UI. В том числе, задачу рендеринга пользовательского интерфеса, формирование стилей и контента, можно переложить на сервер, а можно статично задать на клиенте.

Подробный обзор выбранных архитектурных решений, подходов реализации и технологий представлен во второй главе.

1.3 Выводы по главе 1

В первой главе проведён анализ предметной области. Выделены основные характеристики и черты клиент-серверной архитектуры, разобраны основные понятия и приведены основные достоинства и недостатки каждой архитектуры. Также, произведён анализ существующих решений с приведением достоинств и недостатков.

ГЛАВА 2
ПРОЕКТИРОВАНИЕ СИСТЕМЫ РАЗРАБАТЫВАЕМЫХ ПРИЛОЖЕНИЙ

2.1 Этапы разработки системы приложений

Для разработки системы приложения, необходимо разбить данный процесс на этапы и поставить ряд задач для каждого из этапов.

Работу над разработкой системы приложений можно разбить на следующие этапы:

1. Постановка задачи и анализ требований;
2. Проектирование общей архитектуры системы приложений;
   2.1. Проектирование REST-сервера:
      - описание общей архитектуры приложения;
      - проектирование диаграммы использования;
      - проектирование схемы БД;
      - описание технологий, используемых в разработке;
      - описание основных слоёв приложения;
      - диаграмма классов приложения.
   2.2. Проектирование веб-клиента:
      - описание общей архитектуры приложения;
      - пороектирование шаблонов основных экранов приложения;
      - описание технологий, используемых в разработке;
      - описание основных компонентов приложения;
      - диаграмма деятельности некоторых процессов в приложении.
   2.3. Проектирование Android-клиента:

- описание общей архитектуры приложения
- проектирование шаблонов основных экранов приложения
- описание технологий, используемых в разработке
- описание основных компонентов приложения
- диаграмма деятельности некоторых процессов в приложении

3. Реализация системы приложений:
    3.1. Реализация основных модулей приложений;
    3.2. Ad hook тестирование разработанной системы приложений как отдельно, так и во взаимодействии.

2.2 Определение общего функционала приложений

Исходя из выводов, сделанных в конце первой главы, необходимо определить набор функций, которые будут реализованы в системе приложений. Поскольку будет разработано не одно приложение, а целая система, то и функционал будет разделён по принадлежности к определённому приложению.

2.2.1 Общий функционал REST-сервера

Разрабатываемое приложение должно реализовывать базовые функции интернет-магазина. А именно:

- добавление/изменение/удаление продуктов из БД;
- возможность назначения скидок определённым пользователям;
- обеспечение механизмов идентификации, аутентификации и авторизации;
- возможность формирования заказов на основе товаров, которые клиент положил в свою корзину;
- рассылка e-mail сообщений на основе загруженных шаблонов e-mail сообщений для обеспечения информирования клиентов и менеджеров о состояниях заказов или об объявлениях и акциях;

- управление внутренними файловыми ресурсами приложения;
- реализация сервера изображений, используемых в клиентских приложениях.

2.2.2 Общий функционал веб-клиента

Разрабатываемое приложение должно состоять из двух модулей:

- пользовательская часть;
- CMS-часть.

Пользовательская часть приложения предназначена для использования потенциальными клиентами интернет магазина и должны предоставлять возможности:

- регистрация, аутентификация и авторизация на ресурсе;
- просмотр информации об организации;
- просмотр категорий товаров, товаров и их характеристик;
- наполнение корзины;
- оформление заказа;
- контакт с менеджером;
- просмотр и редактирование личной информации в личном кабинете.

CMS-часть предназначена для использования менеджерами и администратором. Контроль доступа к этой секции осуществляется сервером. Обычный, анонимный пользователь или пользователь с недостаточным уровнем доступа, не может попасть в данную секцию приложения. Основные возможности CMS-части:

- просмотр/добавление/изменение/удаление категорий товаров;
- просмотр/добавление/изменение/удаление товаров, а также изменение списка изображение товара;
- просмотр/добавление/изменение информации о зарегистрированных пользователях, а также редактирование размера их скидок;

- формирование и рассылка почтовых сообщений всем клиентам;

2.2.3 Общий функционал Android-клиента

Разрабатываемое мобольное приложение должно использоваться менеджерами организации, поэтому оно должно обеспечивать возможность авторизации пользователя с помощью установленных на удалённом сервере авторизационных данных. Неавторизованный пользователь не должен иметь возможности получить какие-либо данные из приложения, поскольку это может привести к раскрытию коммерческой тайны.

Авторизованные пользователи должны иметь возможность, в зависимости от уровня доступа:

- просмотр и изменение категорий товаров;
- просмотр и изменение полного списка товаров и их детальные страницы;
- просмотр списка заказов, а также детальной информации по каждому из них;
- просмотр, редактирование и добавление пользователей, редактирование их скидок;
- изменение локальной конфигурации приложения;
- формирование отчетов по определённым критериям.

2.3 Проектирование общей архитектуры приложения

Разрабатываемое приложение является приложением-сервером в клиент-серверной архитектуре. Как следствие, для данной архитектуры необходимо использовать технологии, которые способствуют эффективной реализации всех поставленных задач (см. рисунок 2.1).

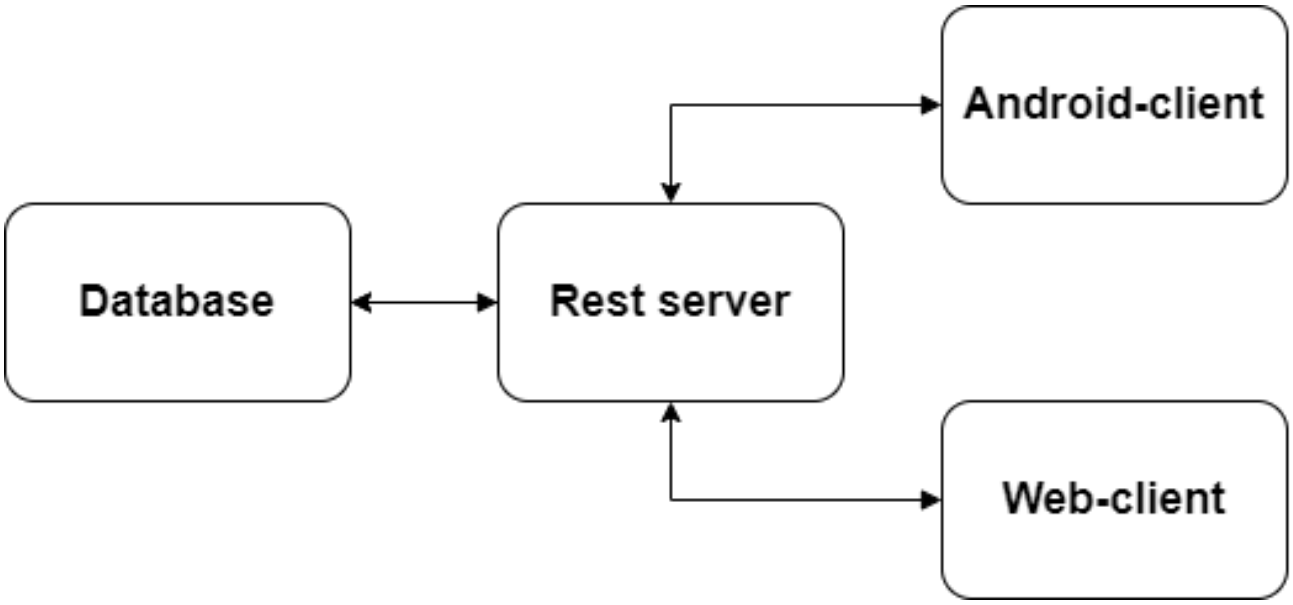


Рисунок 2.1 – Общая архитектура приложения

Для реализации сервера использован REST API подход к реализации архитектуры.

Внутренняя архитектура REST-сервера и веб-клиента следует архитектурному паттерну MVC (Model – View – Controller). В связи с особенностями платформы, Android-клиент реализован при помощи паттерна MVVM (Model – View – ViewModel).

2.3.1 Архитектурный паттерн MVC

MVC – архитектурный паттерн проектирование позволяет разделить приложение на 3 связанные части. Данный паттерн позволяет выделять из больших компонентов части, которые могут быть пере использованы и способствуют параллельной разработке (см. рисунок 2.2).

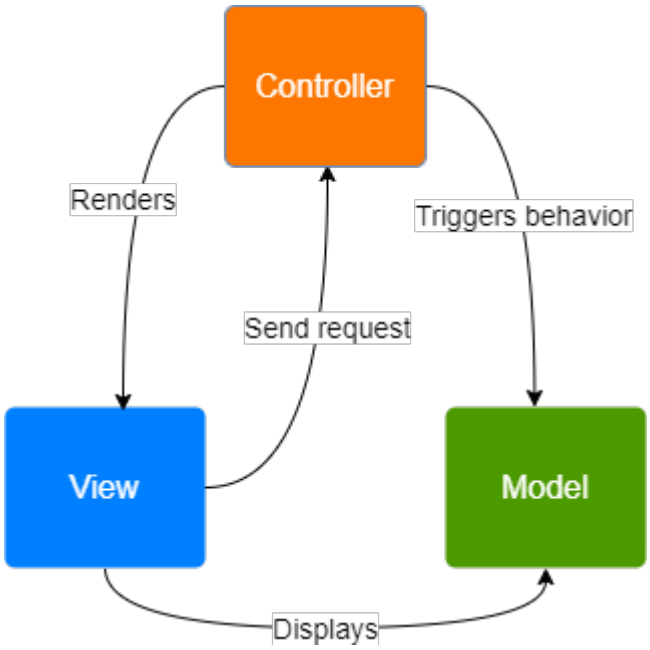


Рисунок 2.2 – Схема архитектурного паттерна MVC

Функциональные слои MVC:

1. Model – представляет собой слой данных и реагирует на инструкции контроллера, изменяя своё состояние;
2. View – отвечает за отображение данных модели пользователю, реагируя на изменение модели;
3. Controller – принимает ввод и преобразует его в инструкции для модели или представления.

2.3.2 Архитектурный паттерн MVVM

MVVC – архитектурный паттерн проектирование позволяет разделить приложение на 3 отдельные части. Данный паттерн позволяет выделять из больших компонентов части, которые могут быть пере использованы и способствуют параллельной разработке (см. рисунок 2.3).

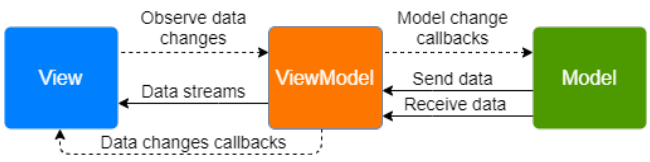


Рисунок 2.3 – Схема архитектурного паттерна MVVM

Функциональные слои MVVM:

1. Model – представляет собой слой данных. Обычно являются структурами или простыми Data классами;
2. View – отвечает за отображение данных модели пользователю, реагируя на изменение модели. Является подписчиком на событие изменения значений свойств или команд, предоставляемых ViewModel. В случае, если в ViewModel изменилось свойство, она оповещает об этом своих подписчиков;
3. ViewModel – содержит Model, преобразованную к View, а также команды, которыми может пользоваться View, чтоб влиять на модель.

2.4 Проектирование диаграммы вариантов использования

Диаграмма вариантов использования представлена на рисунке 2.4.

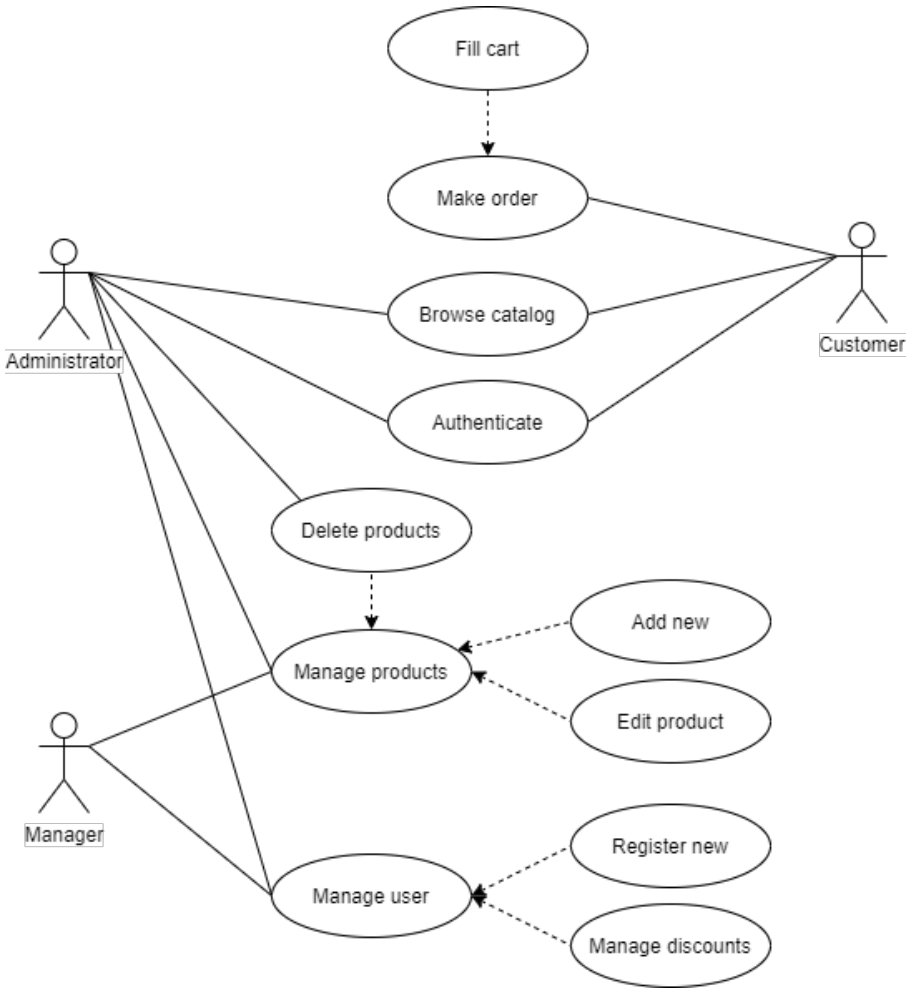


Рисунок 2.4 – Диаграмма вариантов использования

Согласно данной диаграмме, выделяется 3 роли:

1. Клиент – обычный пользователь сайта. Имеет возможность идентификации и аутентификации, просмотра каталога, наполнения корзины в допустимом количестве и оформление заказа.
2. Менеджер – пользователь, занимающийся управлением контентом и актуализацией информации на сайте. Имеет возможности по авторизации со своими пользовательскими данными, управления товарами (добавление/редактирование параметров), пользователями (регистрация новых/редактирование скидок).
3. Администратор – пользователь с неограниченными возможностями по управлению содержимым базы данных.

2.5 Проектирование БД

Спроектированная диаграмма БД представлена в приложении 1.

На данной диаграмме, можно выделить несколько видов связей между таблицами:

1. «Один-ко-многим» – примером такой связи может выступать связь между сущностями ”Категория” и ”Товар”, поскольку, одновременно товар может находиться только в одной категории.
2. «Многие-ко-многим» – реализуется путём применения связей один-ко-многим через промежуточную таблицу. Так, например, сущность ”Товар” связана связью многие-ко-многим с сущностью ”Цвет”, поскольку у каждого товара может быть множество цветов, так и каждый из цветов может быть связан со множеством товаров.
3. «Без связи» – технические, узконаправленные таблицы. К ним относятся таблицы предназначенные для хранения информации, необходимой для идентификации и аутентификации пользователей в системе, а также таблица, содержащая заранее определённые шаблоны почтовых сообщений с темами почтовых сообщений.

2.6 Проектирование макетов основных экранов приложения

2.6.1 Мокапы веб-клиента

Шаблон главной страницы представлен на рисунке 2.5.

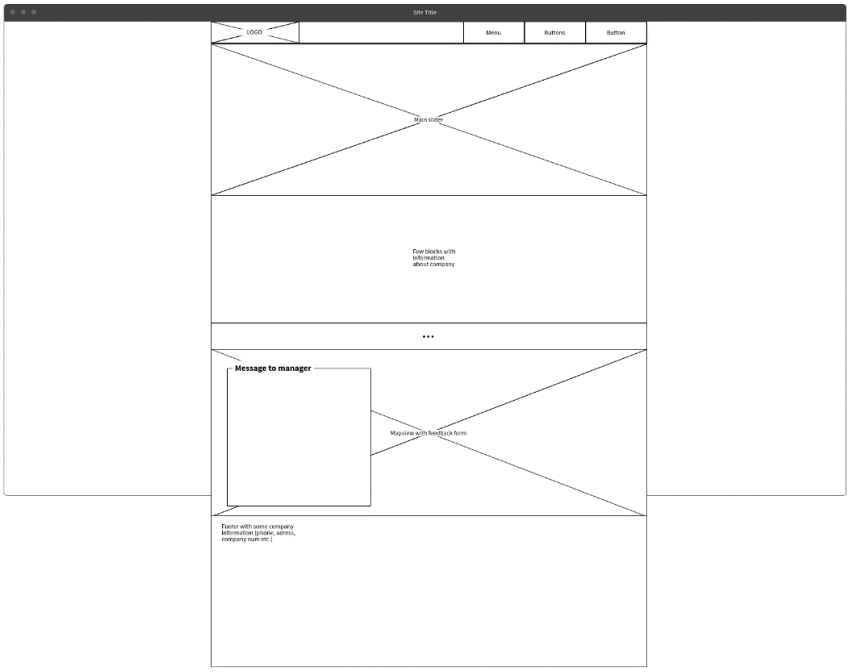


Рисунок 2.5 – Шаблон главной страницы приложения

Тут можно выделить несколько элементов:

- навигационное меню, общее для всех экранов приложения;
- несколько блоков содержащих в себе краткую информацию об организации;
- блок с формой обратной связи и картой, по которой можно определить расположение офиса организации.

Шаблон страницы каталога представлен на рисунке 2.6.

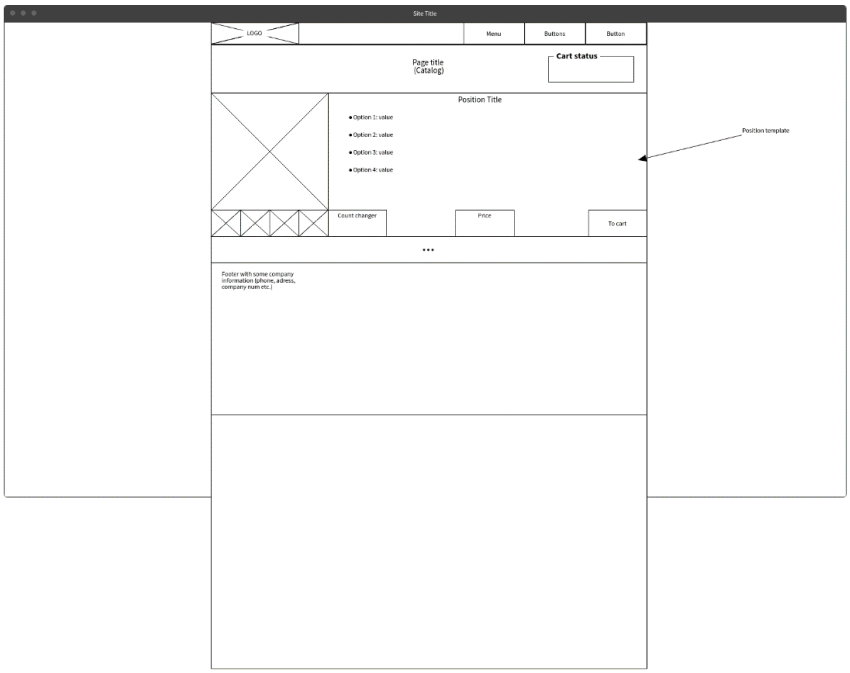


Рисунок 2.6 – Шаблон страницы каталога приложения

Компонент товара в каталоге представлен блоком состоящим из:

- галереи изображений товара;
- блока основной информации о товаре (артикул, заголовок, характеристики, цена);
- кнопка добавления в корзину.

Шаблон страницы оформления заказа и корзины представлен на рисунке 2.7.

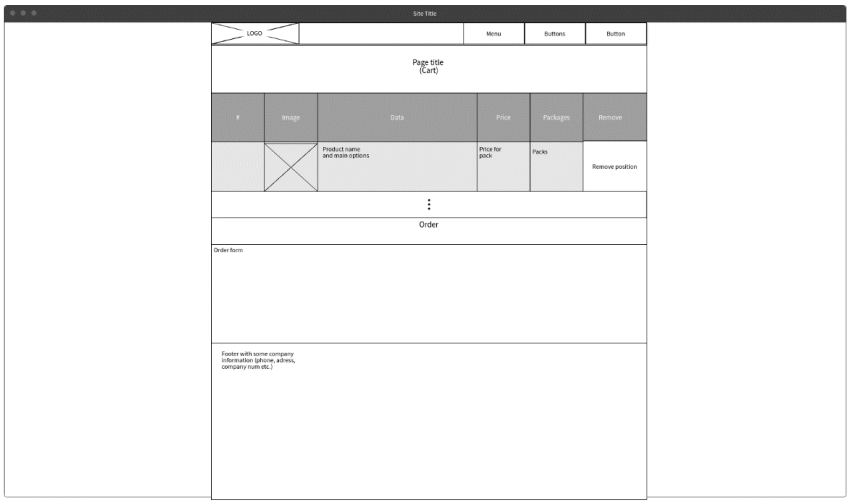


Рисунок 2.7 – Шаблон страницы оформления заказа и корзины

Страница оформления заказа содержит:

- блок просмотра и редактирования заказа:
  - изображение товара;
  - краткая информация о товаре, его цена, кол-во упаковок;
  - переключатель количества упаковок в корзине;
  - кнопка удаления позиции из корзины.
- форму оформления заказа.

2.6.2 Мокапы Android-клиента

Шаблон страницы авторизации в приложении представлен на рисунке 2.8.

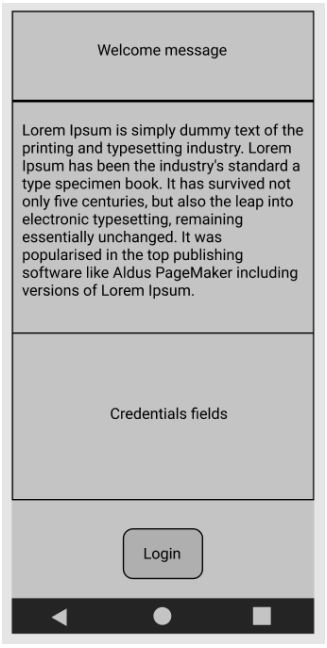


Рисунок 2.8 – Шаблон страницы авторизации приложения

Шаблон страницы каталога представлен на рисунке 2.9.

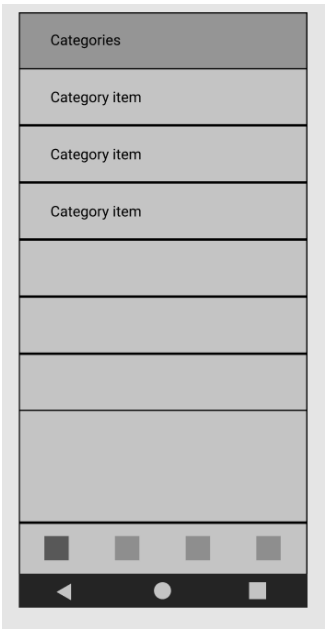


Рисунок 2.9 – Шаблон страницы каталога приложения

Шаблон детальной страницы товара представлен на рисунке 2.10.

Рисунок 2.10 – Шаблон детальной страницы товара

2.7 Описание технологий, используемых в разработке

2.7.1 Технологии, использующиеся при разработке REST-сервера

В качестве языка программирования использован Kotlin.

Kotlin – статически типизированный язык програмиирования, работающий поверх JVM и разрабатываемый компанией JetBrains. Имеет возможность компиляции в Javascript, а также ряд других платформ, через инфраструктуру LLVM. Авторы языка, ставили целью создание более лаконичного и типобезопасного, чем Java и более простого чем Scala языка.[8]

К достоинствам относят:

- лаконичность языка;
- возможность создания расширений для типов, именнованые аргументы и ряд других фич, которые относят к разряду “синтаксического сахара”;
- Kotlin официально поддерживается Google;
- полностью совместим с Java;
- при работающем проекте на Java, имеется возможность не переписывать всё на Kotlin, а лишь дописывать новый функционал, без нарушения

работы в продукте.

К недостаткам можно отнести, достаточно малое сообщество разработчиков, однако оно постоянно расширяется.

Основным фрейворком является Spring Boot,который является упрощенной версией фрейворка Spring.

Spring – один из наиболее популярных фрейворков для разработки приложений для Java (на текущий момент заявлено, что Spring полностью совместим с Kotlin). К основным особенностям фреймворка относят встроенная поддержка DI, которая позволяет придерживаться принципа IoC. Spring помогает свободно разрабатывать полноценные приложения, которые достаточно просто покрываются юнит-тестами.

Spring boot – является упрощенной версией Spring фреймворка. Spring boot позволяет взять на себя часть рутины связанной с конфигурацией проекта.

Spring security и Spring oauth2 – позволяют контролировать доступ к методам приложения, а также позволяет производить авторизацию и регистрацию пользователей.

Spring Data JPA – реализует слой доступа к данным и призван значительно упростить реализацию слоя доступа к данным, сократив усилия на этом этапе и направив в области, которые действительно необходимы. Достоинства:

- поддержка репозиториев, основанных на Spring и JPA;
- поддержка типобезопасных JPA запросов;
- прозрачный аудит для доменных классов;
- поддержка разбивки на страницы;
- возможность интеграции собственного кода для доступа к данным.

Для сборки проекта и управления зависимостями использован Gradle.

Gradle – открытая система для автоматизации сборки проектов. Поддерживает инкрементальную сборку и может определять, какая часть древа была обновлена. Одним из крупнейших преимуществ Gradle по сравнению с другими системами сборки(Maven, Ant и т.д.) является общая

гибкость в настройках сборки и каталогов, без необходимости следовать ограничениям системы сборки.

Для написания Unit-тестов использована библиотека JUnit, которая является библиотекой для модульного тестирования ПО. Изначально данная библиотека была разработана для Java языка. Однако Kotlin полностью совместим с Java, поэтому JUnit может использоваться и для написания тестов для языка Kotlin.

Для общей гибкости при написании тестов использованы библиотеки Mockk и Assertj.

В качестве БД использована реляционная БД MySQL. В реляционной БД данные хрянятся в таблицах. Взаимосвязанные данные могут группироваться в таблицы, а также между таблицами могут быть установлены взаимоотношения. К безусловным достоинствам данной БД является контроль доступа, масштабируемость.

2.7.2 Технологии, использующиеся при разработке веб-клиента

Для вёрстки веб-страниц использован язык разметки HTML. Для предания страницам дизайна, использован CSS.

HTML (от англ. HyperText Markup Language – «язык гипертекстовой разметки») – стандартизированный язык разметки документов во Всемирной паутине. Большинство веб-страниц содержат описание разметки на языке HTML (или XHTML). Язык HTML интерпретируется браузерами; полученный в результате интерпретации форматированный текст отображается на экране монитора компьютера или мобильного устройства. Текстовые документы, содержащие разметку на языке HTML (такие документы традиционно имеют расширение .html или .htm), обрабатываются веб-браузерам, которые отображают документ в его форматированном виде, предоставляя пользователю удобный интерфейс для запроса веб-страниц, их просмотра

(и вывода на иные внешние устройства) и, при необходимости, отправки введённых пользователем данных на сервер.[9]

CSS (от англ. Cascading Style Sheets – каскадные таблицы стилей) – формальный язык описания внешнего вида документа, написанного с использованием языка разметки. Преимущественно используется как средство описания, оформления внешнего вида веб-страниц, написанных с помощью языков разметки HTML и XHTML, но может также применяться к любым XML-документам, например, к SVG или XUL. CSS используется создателями веб-страниц для задания цветов, шрифтов, расположения отдельных блоков и других аспектов представления внешнего вида этих веб-страниц. Основной целью разработки CSS являлось разделение описания логической структуры веб-страницы (которое производится с помощью HTML или других языков разметки) от описания внешнего вида этой веб-страницы (которое теперь производится с помощью формального языка CSS).[9]

Фрейворк для веб-приложения – Angular.

Angular – это открытая и свободная платформа для разработки веб-приложений, написанная на языке TypeScript, разрабатываемая командой из компании Google, а также сообществом разработчиков из различных компаний. Предназначена для разработки одностраничных приложений. Цель использования — расширение браузерных приложений на основе MVC-шаблона, а также упрощение тестирования и разработки.

Фреймворк работает с HTML, содержащим дополнительные пользовательские атрибуты, которые описываются директивами, и связывает ввод или вывод области страницы с моделью, представляющей собой обычные переменные JavaScript. Значения этих переменных задаются вручную или извлекаются из статических или динамических JSON-данных.

Двустороннее связывание данных в Angular является наиболее примечательной особенностью, и уменьшает количество кода, освобождая сервер от работы с шаблонами. Вместо этого, шаблоны отображаются

как обычный HTML, наполненный данными, содержащимися в области видимости, определённой в модели. Специальный сервис в Angular следит за изменениями в модели и изменяет раздел HTML-выражения в представлении через контроллер. Кроме того, любые изменения в представлении отражаются в модели. Это позволяет обойти необходимость манипулирования DOM и облегчает инициализацию и прототипирование веб-приложений.[10]

TypeScript — язык программирования, представленный Microsoft и позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript. TypeScript является обратно совместимым с JavaScript и компилируется в последний. Фактически, после компиляции программу на TypeScript можно выполнять в любом современном браузере или использовать совместно с серверной платформой Node.js. Код экспериментального компилятора, транслирующего TypeScript в JavaScript, распространяется под лицензией Apache. Его разработка ведётся в публичном репозитории через сервис GitHub. TypeScript отличается от JavaScript возможностью явного статического назначения типов, поддержкой использования полноценных классов (как в традиционных объектно-ориентированных языках), а также поддержкой подключения модулей, что призвано повысить скорость разработки, облегчить читаемость, рефакторинг и повторное использование кода, помочь осуществлять поиск ошибок на этапе разработки и компиляции, и, возможно, ускорить выполнение программ [11].

Angular Material состоит из набора предустановленных компонентов Angular. В отличие от Bootstrap, предоставляющего компоненты, которые вы можете использовать любым способом, Anglate Material стремится обеспечить расширенный и последовательный пользовательский интерфейс. В то же время он дает возможность контролировать, как ведут себя разные компоненты.

Material Design – это язык дизайна для веб и мобильных приложений, который был разработан Google. Material Design упрощает разработчикам настройку UI, сохраняя при этом удобный интерфейс приложений.

2.7.3 Технологии, использующиеся при разработке Android-клиента

Для разработки нативного Android-приложения использован Android-фреймворк использован язык программирования Kotlin.

В разработке использованы элементы из Android Jetpack Architecture Components:

- LiveData – хранилище данных, работающее по принципу паттерна Observer, которое умеет определять активность подписчика;
- LifeCycle – компонент для удобной работы с LifeCycle Activity;
- Android Ktx – функции расширения для стандартной библиотеки Android;
- Navigation – компонент облегчающий навигацию между фрагментами Android приложения;
- Room – ORM система для SQLLite;
- ViewModel – компонент позволяющий корректно обрабатывать состояние фрагмента или активити при изменении состояния (например, при повороте).

В качестве DI фреймворка выступает Koin. Koin – небольшая библиотека для внедрения зависимостей. В отличии от большей части подобных библиотек, Koin не использует кодогенерацию, проксировани или итроинспецкию. Из дополнительных плюсов, Koin использует DSL и функционал языка Kotlin. Подразумевается использования с Kotlin, однако, Java тоже может работать вместе с Koin.

OkHttp использован для реализации возможности выполнения сетевых запросов и сетевого взаимодействия. Эта библиотека обладает полным функционалом для работы с любым REST API, легко тестируется и настаивается.

2.8 Описание основных компонентов приложений

2.8.1 Основные компоненты веб-клиента

К основным страницам разрабатываемого приложения относятся:

- главная страница;
- каталог;
- корзина/Форма оформления заказа;
- страница “Товары” CMS-части;
- страница “Пользователи” CMS-части.

Разберём основной функционал, который должны предоставлять данные страницы.

Главная страница – является отправной точкой для пользователя и содержит основную информацию об организации, владеющей интернет магазином. По сути является Landing-page.

Каталог – содержит список товаров, по категориям, которые отсортированы по наличию и цене. Каждый товар обладает своим рядом характеристик, а также изображениями. Некоторые товары могут обладать цветами, в таком случае, в корзину складывается не просто товар, а еще и его цвет. Изображения каждого из товаров переключается с заданным интервалом. В случае, отсутствия изображений, блок с изображениями заменяется на fallback-image.

Корзина/Форма оформления заказа – предоставляет возможность изменить кол-во товара и их список, которые будут использованы при оформлении заказа. Каждый товар обладает рядом осноных характеристик и одним изображением. Цена за позицию должна изменяться на лету, в зависимости от кол-ва товаров в корзине. Форма оформления заказа должна поддерживать валидацию введённых данных, перед отправкой запроса на сервер. В случае успешного оформления заказа, происходит переадресация на

главную страницу приложения и очистки локальной корзины.

Страница “Товары” CMS-части – страница администраторской части приложения, которая доступна только пользователями с уровнем доступа Manager и выше. Контроль доступа осуществляется сервером. На данной странице есть возможность просмотра/добавления/редактирования и изменения категорий товаров и товаров. Товары сгруппированы по категориям. Имеется возможность быстрой установки информации о том, что товар отсутствует на складе или удалён. Кроме того, присутствует возможность изменения информации о товаре и его изображения на специальной форме. Для добавления товара используется отдельная форма.

2.8.2 Основные компоненты Android-клиента

К основным страницам разрабатываемого приложения относятся:

- страница авторизации;
- список категорий и их продуктов;
- страница детальной информации о товаре;
- список заказов и информация о них;
- список зарегистрированных пользователей, с возможностью детального просмотра информации, а также удалении/добавления.

Разберём основной функционал, который должны предоставлять данные страницы.

Контроль доступа осуществляется со стороны сервера.

Страница авторизации – является отправной точкой для пользователя и содержит небольшое приветственное сообщение и поля для ввода авторотационных данных.

Список категорий и их продуктов содержит список товаров, по категориям. Каждый товар обладает своим рядом характеристик, а также сопровождается изображениями. Некоторые товары могут обладать

цветами. Изображения каждого из товаров переключются с заданным интервалом. В случае, отсутствия изображений, блок с изображениями заменяется на fallback-image. На данной странице есть возможность просмотра/добавления/редактирования и изменения категорий товаров и товаров. Товары сгруппированы по категориям. Имеется возможность быстрой установки информации о том, что товар отсутствует на складе или удалён. Кроме того, присутствует возможность изменения информации о товаре и его изображения на специальной форме. Для добавления товара используется отдельная форма.

2.9 Диаграмма деятельности некоторых функций приложения

2.9.1 Диаграмма деятельности веб-клиента

Диаграмма деятельности веб-клиента представлена на рисунке 2.11

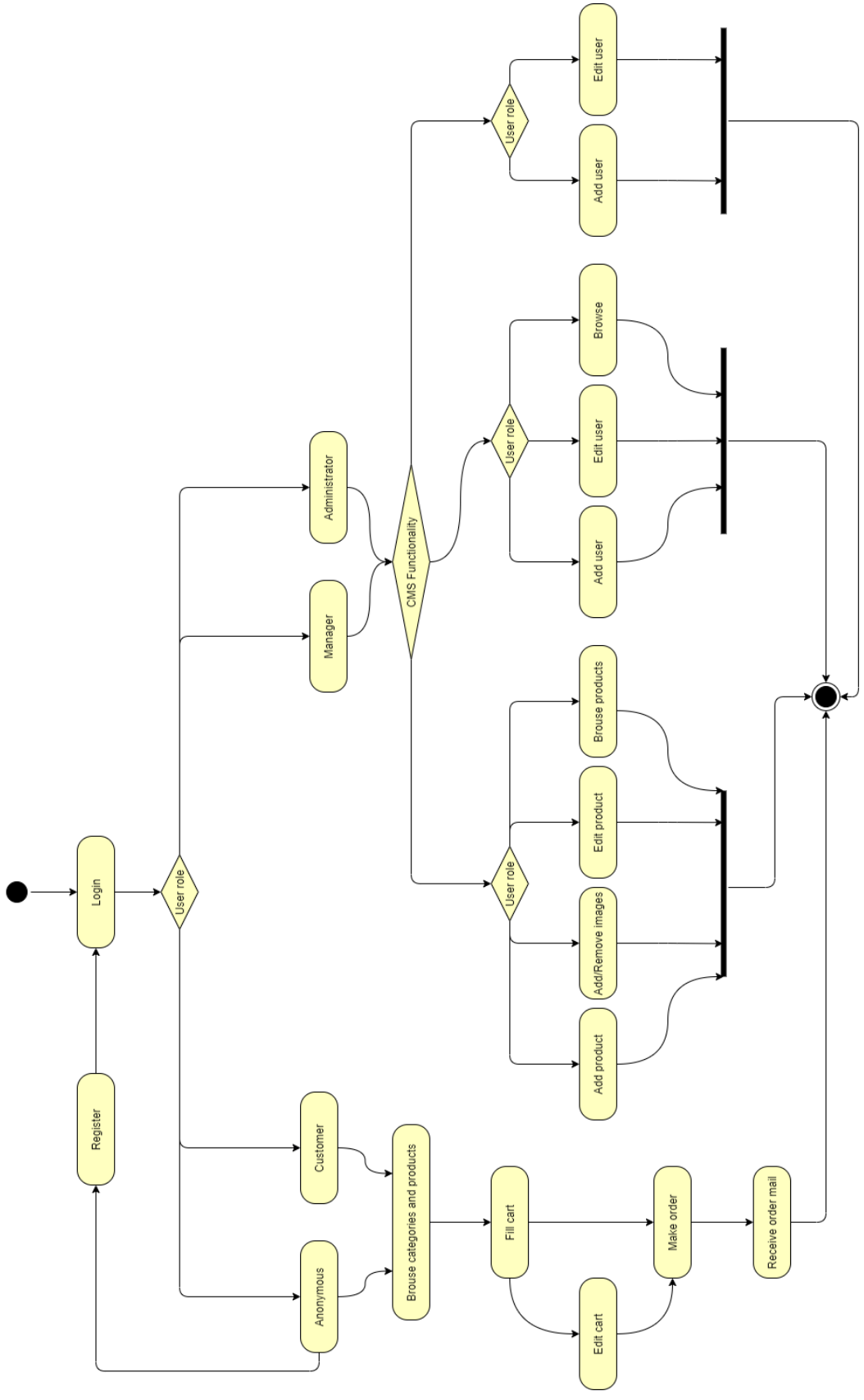


Рисунок 2.11 – Диаграмма деятельности веб-клиента

Согласно представленной диаграмме, после авторизации пользователя в приложении, в зависимости от роли, которая закреплена за данным пользователем, изменяется список функционала, который доступен пользователю. Так, например, анонимный пользователь, имеет возможность зарегистрироваться/авторизоваться в приложении, в последствии получив одну из ролей, которая закреплена за данным пользователем. Помимо авторизации и регистрации, пользователь имеет возможность просматривать каталог товаров, наполнять корзину и делать заказ.

В случае, если пользователь обладает уровенем доступа выше уровня ”Менеджер”, включительно, для него открывается доступ в CMS часть сайта, в которой есть возможность редактировать список товаров, категорий, пользователей и т.д.

2.9.2 Диаграмма деятельности Android-клиента

Диаграмма деятельности Android-клиента представлена на рисунке 2.12

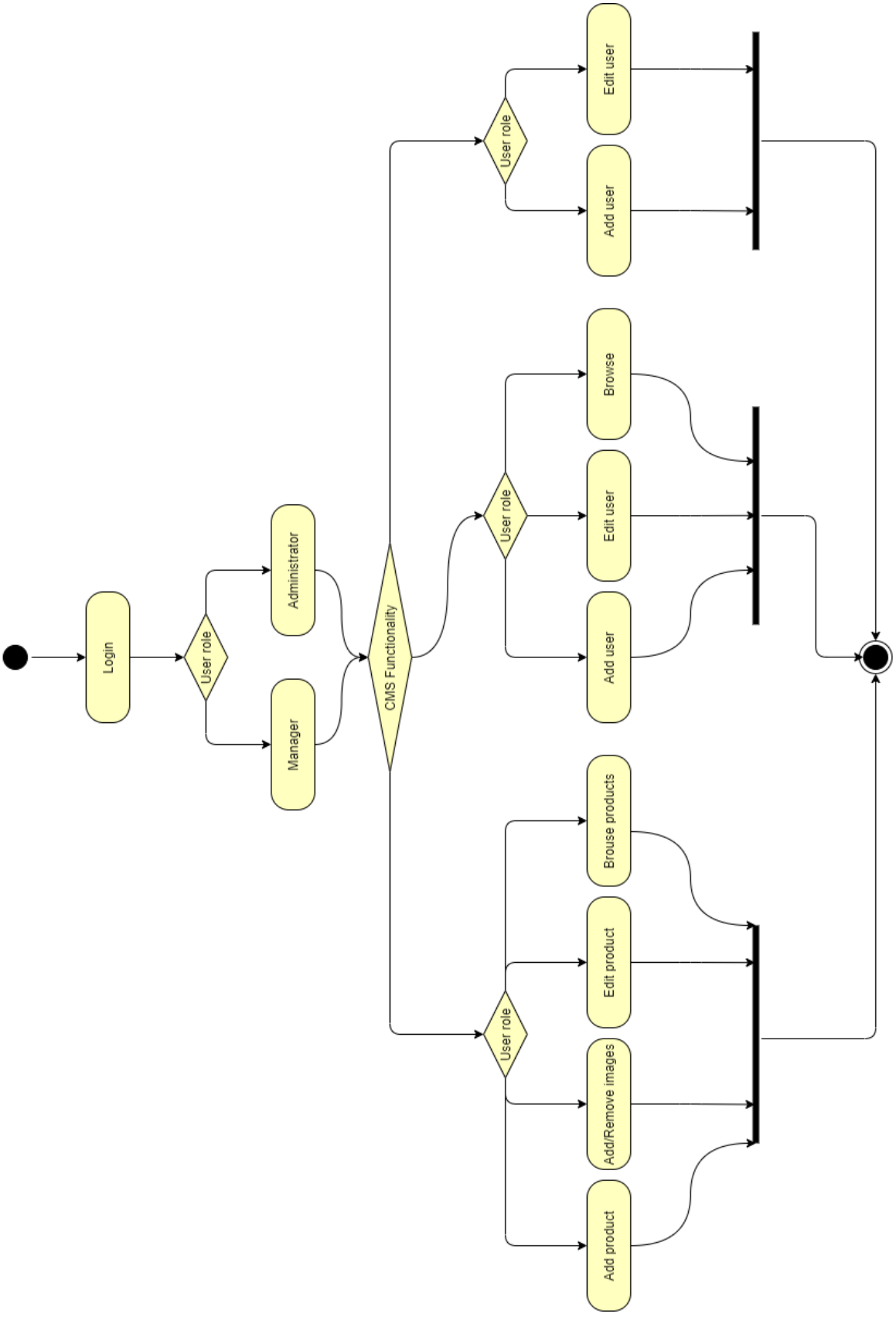


Рисунок 2.12 – Диаграмма деятельности Android-клиента

Диаграмма деятельности Android-клиента имеет схожую структуру и суть с диаграммой веб-клиента, за исключением невозможности работы неавторизованным пользователем и невозможностью пользоваться функционалом предназначенным для простого пользователя приложения (например, складывать товары в корзину или оформлять заказ). Однако, пользователю Android-приложения доступен весь функционал, доступный менеджерам и администраторам CMS-части веб-клиента.

2.10 Вывод по главе 2

Во второй главе описаны основные этапы разработки приложения, функционал, а также перечислены основные технологии и фреймворки, которые будут использованы в процессе разработки. Представлена диаграмма вариантов использования, схема БД, диаграммы деятельности, макеты основных экранов и сформированы требования для каждого из экранов разрабатываемых клиентских приложений.

ГЛАВА 3
РЕАЛИЗАЦИЯ СИСТЕМЫ ПРИЛОЖЕНИЙ ИНТЕРНЕТ-МАГАЗИНА

3.1 Реализация приложения REST-сервера

3.1.1 Описание архитектурных решений

Для реализации выбран архитектурный паттерн MVC. Данный паттерн позволяет разделить данные, представление и бизнес-логику.

Все классы разделены на 3 слоя:

1. Данные – в этот слой входят все классы из пакетов model и repository.
2. Сервис – в этот слоя входят классы из пакета service. На этом уровне выполняется основная бизнес-логика приложения. Сервис существует для каждой значимой сущности и абстрагирован от других сущностей.
3. Контроллер – в этот слой входят классы из пакета controller. На данном слое происходит json-сериализация с помощью встроенного в Spring-framework сериализатора Jackson, обработка ошибок и формирования ответов клиенту. Для получения/отправки данных используется шаблон проектирования DTO (Data Transfer Object).

Отдельными модулями приложения являются пакеты:

1. Utils – пакет в котором хранятся общие утилиты, необходимые приложению, а также поддерживаемые Kotlin-ом расширяющие функции.
2. Config – пакет в котором производится конфигурация Spring фреймворка. А также создаваемые для Spring DI – компоненты. Примеры конфигурации бинов будут приведены в приложении.

3.1.2 Описание основных аннотаций

Для маппинга Kotlin data классов, используемых в БД используются JPA аннотации.

@Entity – используется для сообщения Spring о том, что класс является сущностью используемой в БД.

Т.к. для каждого класса, используемого в БД необходим, конструктор по умолчанию (без параметров), а kotlin-data классы не поддерживают его, в приложении используется kotlin-noarg gradle плагин, который генерирует конструктор по умолчанию для всех data классов.

@Table – используется для указания имени таблицы, которое будет использовано при обращении.

@get: – является решением, для того чтоб размещать аннотации над соответствующим геттером(т.к. в Kotlin геттеры не пишутся и генерируются автоматически).

@Column – используется для указания имени, а также некоторого списка характеристик колонки таблицы (например nullable) определённого поля класса.

Если в классе есть поля объектного типа, то должна быть указана связь между таблицами (@OneToMany, @ManyToOne, @ManyToMany), а так же аннотация для указания, по какому полю производить связь (@JoinColumn).

@Controller, @RestController, @Service, @Repository – используются для указания Spring о том, к какому типу компонента относится тот или иной класс.

@Bean – сообщает Spring о том, что объект является Spring-Bean.

@Autowired – сообщает Spring о том, что реализацию данного поля нужно найти среди Spring-Beanов.

@Configuration – указывает на то, что класс является конфигурационным.

Связка @JsonManagedReferense/@JsonBackReference является аннотациями Jackson-сериализатора и служат для того, чтоб избежать

рекурсивной десериализации объектов, которые ссылаются друг на друга.

@JsonIgnore – служит для того, чтоб указать Jackson-сериализатору то, что данное поле следует игнорировать пре сериализации.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @RequestMapping – аннотации сообщающие диспатчеру о том, по какому url адресу, данный метод должен обрабатывать запросы.

3.1.3 Описание процесса авторизации

Для контроля доступа и возможности назначения персональных скидок, в системе реализована возможность авторизации. Авторизация происходит по протоколу OAuth2

Преимущества OAuth2:

- Клиент может быть уверен в том, что несанкционированный доступ к его личным данным невозможен. Не владея логином и паролем пользователя, приложение может выполнять только ограниченный ряд действий.
- Не нужно заботиться об обеспечении конфиденциальности логина и пароля. Т.к. логин и пароль не передаются приложению и как следствие, не могут попасть в руки злоумышленников.

Результатом авторизации является получение access token – некий ключ (хешированая строка) предъявление которого является доступом к защищенным ресурсам. Самым простым способом передачи является его указание в заголовках вместе с запросом.

3.1.4 Описание работы почтового клиента с генерацией шаблонов

Для отправки e-mail сообщений используется Java-класс JavaMailSender, который сконфигурирован как Spring Bean и доступен для Dependecy Injection. Для конфигурации данного класса ему передаются список параметов (как

например логий и пароль от SMTP сервера, который будет рассылать сообщения)

На текущий момент сообщения отправляются при оформлении заказа. Т.е. когда приходит запрос на регистрацию заказа, заказ сначала добавляется в БД, в случае успешного добавления, из БД получается необходимый шаблон почтового сообщения, в который вставляются данные заказа. Для вставки корректных данных, в шаблоне предусмотрены специальные метки заданные регулярным выражением: ”<\\[([^\\]%]*)\\]>”

3.1.5 Unit-тестирование

Unit-тестирование – процесс позволяющий проверить на корректность отдельные модули исходного кода программы, с соответсвующиеми управляющими данными, процедурами использования и обработки. Идея заключается в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить работоспособность кода и не привело ли изменение к регрессии.

Для упрощения написания тестов, в приложении используются библиотеки Mockk и AssertJ.

Mockk – это простая Kotlin библиотека для создания объектов заглушек, которые не несут в себе никакой логики, однако используются для симуляции поведения объектов с определёнными условиями. Т.к. эта библиотека работает по приципу наследования от мокируемого объекта, а в Kotlin все классы являются по умолчанию ненаследуемыми, необходимо использовать allopen graddle плагин, который делает все классы открытыми по умолчанию.

AssertJ – библиотека, которая предоставляет удобный интерфейс для написания тестовых сравнений и главной целью ставит улучшение читаемости тестового кода, а также повышения удобности отладки тестов.

В текущей системе, тестами покрыты все нетривиальные методы Сервис-слоя, а также сериализаторы и парсер почтовых сообщений.

3.1.6 Сборка проекта

Для сборки проекта используется система автоматической сборки Gradlle.

Данный сборщик поддерживает написание build-скриптов на языке Kotlin, Groovy. Имеется возможность тонкой настройки скриптов сборки, а также дополнительных задач, которые будут выполнены перед сборкой.

Кроме того, есть возможность скачивания зависимостей из maven-repository сервисов и поддержка плагинов (в данном приложении, например были использованы плагины allopen и noargconstr, для обеспечения совместимости языка Kotlin с некоторыми библиотеками или фреймворками. Также, поддержка инкрементальной сборки и отслеживание изменённого кода, может значительно сократить время сборки (особенно это видно на больших проектах).

3.2 Реализация приложения веб-клиента

3.2.1 Описание архитектурных решений

Для реализации выбран архитектурный паттерн MVC. Данный паттерн позволяет разделить данные, представление и бизнес-логику.

Все файлы приложения были разделены на 3 слоя:

1. Данные – в этот слой входят все файлы, которые отвечающие за представление структур данных, которые используются в приложении. Все данные, которые приложение получает с сервера или с локального хранилища представлены в виде интерфейсов данного слоя.
2. Сервис – на этом уровне выполняется основная бизнес-логика приложения и запросы к серверу. Сервис существует для каждого значимого функционала значимой сущности и соответствует SPR

(Single Responsibility Principle).

3. Контроллер – на данном слое происходит обработка ошибок и биндинг данных в view, а также происходит управление состоянием view в зависимости от существующих данных, либо статуса загрузки данных с сервера.

Отдельными модулями приложения являются:

- Pipes – содержит классы, которые занимаются форматирование данных при отображении. Например, CountPipe, PricePipe
- Utils – содержит классы, утилиты и икапсулированные обёртки вокруг библиотек, настроенные для использования в реализуемом приложении.
- Routing – содержит всю логику и весь маппинг возможных переходов по приложению.

3.2.2 Описание основных сервисов

В приложении используется аутентификация на сервере по технологии OAuth2, поэтому должна быть реализована логика, которая может быть легко встраиваемой в любой компонент при помощи DI. Кроме того, должны быть механизмы перезапроса access_token’a, при наличии refresh_token’a, в случае истечения срока его действия. Для этого был реализован AuthorizationService, листинг которого можно увидеть в приложении 2. Данный сервис инкапсулирует в себе логику для контроля авторизационных процессов. Поскольку данный класс реализует в себе HttpInterceptor интерфейс, он может быть добавлен как перехватчик к любому исходящему запросу и выполнять необходимую логику перезапроса токена, при наличии refresh_token’a и получении 403 ошибки при выполнении запроса и добавления токенов в заголовки запроса, при их наличии. Все токены хранятся в local storage браузера.

CartService занимается контролем за состоянием корзины, а также её управлением.

На каждый из контроллеров сервера, реализованы свои сервисы.

Основная их задача заключается в выполнении запроса к серверу и возвращение подписки на UI. При помощи данного callback'а имеется возможность выполнения запросов в сеть без блокирования UI потока.

Пример реализации MgrProductService можно найти в приложении 3.

Пример реализации подписки на получаемый результат от сервиса представлен на листинге 1.

```
1  private loadProducts(id: number) {
2      this.productService.getByCategoryId(id).subscribe(
3      products => {
4          this.tableConfig.source = new MatTableDataSource<Product>(products);
5
6          setTimeout(() => {
7              ProductComponent.scrollToView(this.productActionsSubSection);
8              this.tableConfig.source.sort = this.sort;
9              this.tableConfig.source.paginator = this.paginator;
10         });
11     }, error => {
12         this.processError(error);
13     }, () => this.setLoading(false)
14     )
15 }
```

Листинг 1 – Пример реализации подписки на ожидаемый результат от сервера

3.2.3 Описание принципов построения пользовательского интерфейса

В Angular пользовательский интерфейс состоит из легко встраиваемых компонентов. Каждый компонент создаётся разработчиком и может управлеяет отображением представления на экране. Для создания компонента необходимо импортировать функцию декоратора @Component из библиотеки @angular/core. Данный декоратор позволяет идентифицировать класс как компонент.

Декоратор в качестве параметра принимает объект с конфигурацией, которая указывает фреймворку, как работать с компонентом и его представлением. С помощью свойства template, шаблон представляет часть HTML разметки с вставкой кода Angular. Фактически, шаблон и является

представлением, которым пользователь управляет при работе с приложением. Каждый компонент должен обладать одним шаблоном. Свойство selector определяет селектор CSS. В элемент с этим селектором Angular будет добавлять представление компонента.

Некоторые элементы форм клиентской части и вся CMS-часть приложения используют Angular Material Components.

Для обеспечения адаптивности приложения используется CSS-Grid Layout. Данный подход позволяет менять расположение grid элементов не меняя сам HTML. К основным понятиям CSS Grid относят:

- Grid container – набор пересекающихся горизонтальных и вертикальных grid линий, которые делят пространство контейнера на области, в которые могут быть помещены grid элементы.
- Grid lines – это горизонтальные и вертикальные разделители grid контейнера. Эти линии находятся по обе стороны от столбца или строки. Разработчик может задать для данного элемента имя или числовой индекс, которые может использовать дальше в стилях. Нумерация начинается с единицы. Важный нюанс, данный элемент восприимчив к режиму написания, который используется на вашем ресурсе. Например, вы используете Арабский язык или любой другой язык у которого режим написания справа налево, то нумерация линий будет начинаться с правой стороны.
- Grid track – это пространство между двумя смежными grid линиями, вертикальными или горизонтальными.
- Grid cell – это наименьшая неделимая единица grid контейнера на которую можно ссылаться при позиционировании grid элементов. Образуется на пересечении grid строки и grid колонки.
- Grid area – это пространство внутри grid контейнера, в которое может быть помещен один или больше grid элементов. Этот элемент может состоять из одной или более grid ячеек.

Каждый элемент тесно связан друг с другом и отвечает за определенную часть grid контейнера.

Пример HTML для ProductAdd компонента можно найти в приложении 4. CSS для этого компонента находится в приложении 5.

3.2.4 Описание основных сторонних библиотек

Основные сторонние библиотеки, используемые в приложении:

- Ngx-gallery – библиотека предоставляющая компонент для простой реализации автоматической галлереи изображений, обладающая рядом дополнительных функций. Используется на странице каталога товаров.
- Ngx-infinite-scroll – библиотека предоставляющая возможность порционной загрузки данных по мере приближения к концу страницы. Используется на странице каталога товаров.
- RxJs – библиотека для обеспечения возможности реактивного программирования.
- Angular4-carousel – библиотека предоставляющая слайдер компонент используемый на главной странице приложения.
- Angular-notifier – библиотека предоставляющая настраиваемые всплывающие уведомления, которые используются в ответ на действия пользователя, по всей клиентской части приложения.
- Angular-2-local-storage – библиотека предоставляющая абстрактную обёртку вокруг local-storage, которая инкапсулирует всю логику работы с ним и предоставляет удобный интерфейс разработчику.

3.2.5 Сборка и структура проекта

Для разработки приложения использовался Angular CLI – интерфейс командной строки, который позволяет быстро создавать проекты, добавлять файлы и выполнять множество определённых задач, таких как тестирование,

сборка и развёртывание. Для корректной работы Angular CLI, необходимо чтоб были установлены Node.js и npm.

Для запуска веб-сервера, используемого для разработки приложения необходимо выполнить комманду ng serve –open в дирректории Angular приложения. Команда ng serve запускает веб-сервер, а также прослушивает каталог с исходниками приложения и при изменениях в этих исходных файлах пересобирает проект «на лету». Стоит отметить, что в таком режиме проект не сохраняется на диске, он записывается непосредственно в оперативную память. Использование ключа –open (или просто -o) означает, что после сборки проекта, автоматически откроется браузер (по умолчанию выбранный в операционной системе).

Пример структуры angular приложения представлен на листинге 2.

```
.
|-- app
|   |-- app.component.css
|   |-- app.component.html
|   |-- app.component.spec.ts
|   |-- app.component.ts
|   `-- app.module.ts
|-- assets
|-- environments
|   |-- environment.prod.ts
|   `-- environment.ts
|-- favicon.ico
|-- index.html
|-- main.ts
|-- polyfills.ts
|-- styles.css
|-- test.ts
|-- tsconfig.app.json
|-- tsconfig.spec.json
`-- typings.d.ts
```

Листинг 2 – Пример структуры angular приложения

Исходники приложения, располагаются в директории src.

app/app.component.ts,html,css,spec.ts – корневой компонент приложения в который внедряются все остальные компоненты приложения. В нём указан корневой компонент иерархии представления. Сопровождается html-шаблоном, css-стилями и юнит-тестами.

app/app.module.ts – определяет корневой модуль AppModule, в котором определено как собирается приложение.

*assets/** – директория, в которой размещаются файлы ресурсов использующиеся в приложении. После сборки приложения, ресурсы

копируются без изменений.

index.html – главная HTML-страница, которая загружается при посещении пользователем. AngularCLI автоматически добавлет весь JavaScript код и CSS файлы при сборке.

main.ts – главная точка входа Angular приложения. По умолчанию приложение компилируется JIT(Just In Time) компилятором и запускает его в браузере.

Пример структуры корневой директории проекта представлен на Листинге 3.

```
.
|-- README.md
|-- e2e
|-- karma.conf.js
|-- node_modules
|-- package-lock.json
|-- package.json
|-- protractor.conf.js
|-- src
|-- tsconfig.json
`-- tslint.json
```

Листинг 3 – Пример структуры корневой директории проекта

node_modules/ – окружение Node.js создает данную директорию, в которой хранятся все сторонние модули, подключаемые из-вне путём перечисления в package.json.

.angular-cli.json – конфигурационный файл AngularCLI. Посредством этой конфигурации можно установить некоторые из значений сборки по умолчанию, а также конфигурировать список файлов которые будут использованы при сборке проекта.

.editorconfig – конфигурационный файл редактора кода. Специфицирует конфигурацию форматирования текста кода, большинство современных редакторов кода поддерживает конфигурацию полученную из данного файла.

.gitignore – файл контроля версий, содержит список файлов которые надо игнорировать при загрузке не в систему контроля версий Git-репозиторий. К разряду не нужных файлов относятся файлы сгенерированные редактором кода библиотеками кодогенерации.

package.json – конфигурационный файл npm, в нем перечисляются сторонние модули (пакеты) разработчиков, которые использует ваш проект.

tsconfig.json – конфигурация компилятора TypeScript для редактора кода.

tslint.json – конфигурация для статического анализатора TSLint, используется при запуске ng lint.

3.3 Реализация приложения Android-клиента

3.3.1 Описание архитектурных решений

Для реализации выбран архитектурный паттерн MVVM. Данный паттерн позволяет разделить данные, представление и бизнес-логику.

Все файлы приложения разделены по пакетам-фичам (англ. Feature). Каждая из таких фич имеет в себе строгую иерархию классов, которая помогает следовать SPR:

1. Fragment – слой View. Является отображением модели.
2. ViewModel – слой ViewModel. Хранит в себе объект LiveData и изменяет его в зависимости от каких-либо сценариев. Кроме того, содержит в себе один или несколько UseCase. В LiveData хранится объект состояния Фрагмента.
3. State – инкапсулирует в себе данные и состояние View (Например, Loading, DataReady и т.д.).
4. UseCase – класс отвечающий за одно определённое действие. Например, получение списка всех товаров. В UseCase может быть внедрён один или несколько репозиториев. На данном уровне выполняется запуск и контроль корутин
5. Repository – слой отвечающий за получение данных из каких-либо источников (сеть или локальная база данных в зависимости от ситуации). Обычно в себе содержит несколько API-классов, которые инкапсулируют в себе запросы на удалённый сервер и парсинг

полученной модели и несколько dao-классов, которые инкапсулируют в себе получение данных из локальной базы данных

Отдельными пакетами приложения являются:

- Networking – содержит классы и API-интерфейсы, которые инкапсулируют в себе логику сетевых запросов
- Database – содержит классы, API-интерфейсы и модели данных которые инкапсулируют в себе логику запросов в базу данных
- CommonUtils – Небольшие утилитарные классы и функции-расширения

Для использования технологии OAuth2 реализован AuthTokenInterceptor, листинг которого можно найти в приложении 6. Данный сервис инкапсулирует в себе логику для контроля авторизационных процессов. Поскольку данный класс реализует в себе Interceptor интерфейс, он может быть добавлен как перехватчик к любому исходящему запросу и выполнять необходимую логику перезапроса токена, при наличии refresh_token'а и получении 403 ошибки при выполнении запроса и добавления токенов в заголовки запроса, при их наличии. Все токены хранятся в sharedPreferences.

На каждый из контроллеров сервера, реализованы свои API-классы. Основная их задача заключается в выполнении запроса к серверу и возвращении данных на уровень Репозитория для дальнейшей обработки или кеширования.

Для того, чтоб не блокировать UI поток во время выполнения сетевых запросов, используются Kotlin-корутины. Контроль за созданием и переключением контекстов корутин находится на уровнях UseCase-Repository.

Пример реализации можно найти в приложении 7.

На уровне Fragment происходит подписка на изменение состояния ViewModel. Пример реализации подписки на получаемый результат от ViewModel представлен в листинге 4.

```
productsListViewModel.model.observe(viewLifecycleOwner) {
    when (it) {
        is Loading -> onLoading()
        is NoData -> {
            onLoadingStopped()
            list_products.visibility = GONE
            text_no_data.visibility = VISIBLE
        }
        is DataReady -> {
            adapter.cleanAddAll(it.products)
            onLoadingStopped()
            text_no_data.visibility = GONE
            list_products.visibility = VISIBLE
        }
    }
}
```

Листинг 4 – Пример реализации подписки на ViewModel

3.3.2 Описание процесса кэширования данных

Поскольку одним из Одним из требований разрабатываемого приложения является возможность работы без подключения к интернету. Поэтому необходимо разработать механизм кэширования данных полученных в результате HTTP запросов к серверу, чтобы отображать пользователю последнюю полученную с сервера информацию.

Кэширование – один из способов оптимизации приложений. Его суть заключается в сохранении данных, которые были получены тяжеловесной операцией из какого-либо источника, в памяти приложения/базе данных и т.д. для дальнейшего более быстрого его получения и обработки. Кроме того, кеширование позволяет разгрузить нагрузку с сервера, поскольку к нему не будут выполняться запросы каждый раз, а только при истечении актуальности локального кэша, либо при принудительном запросе обновления данных. В контексте мобильного приложения, кэширование позволяет создать offline режим работы приложения. В таком случае, приложение будет иметь возможность отображать последние закэшированные данные даже без подключения к интернету.

Политики кеширования задаются полем Cache-Control общего заголовка HTTP. HTTP клиент OkHttp предоставляет, позволяющую автоматически котролирвать кэширование запросов. Однако, в рамках дипломной работы,

данная функция не использована и вся логика кеширования HTTP запросов выполнена самостоятельно.

Рассмотрим логику кеширования данных и проверки их актуальности.

Для этого, необходимо создать единую точку входа, через которую будут проходить все запросы клиента на загрузку данных и в случае их устаревания, разрешать выполнение запроса к серверу данных. Таковым местом в приложении будет являться объект HttpRequestManager. Он будет зарегистрован в DI Koin, как single, что гарантирует то, что данный объект будет являться singleton'ом. На вход, метод HttpRequestManager#request получает следующие параметры:

- path – относительный путь, по которому надо выполнить запрос.
- method – HTTP метод с которым будет выполнен запрос
- cacheControl – объект содежащий информацию о том, когда ранее загруженные данные перестанут быть актуальными. В случае, если этот параметр null, кеширование не будет произведено
- queryParams – параметры с которыми надо выполнять запрос

Для сохранения данных о том, когда в последний раз был выполнен какой-либо запрос, в локальной базе данных создана сущность RequestCache. Эта сущность содежит в себе информацию о запросе, параметрах, методе, пути и времени, когда были выполнены все запросы, которые необходимо кешировать. Перед любым запросом к серверу и БД удаляются все неактуальные записи. В случае, если при попытки выполнения запроса к серверу, в таблице RequestCache уже будет содержаться запись об актуальности запроса, то запрос не будет выполнен и данные будут прочитаны из БД. Идентификация запросов выполняется по хэш-коду модели запроса.

Полный листинг HttpRequestManager класса с обработкой кэша расположен в приложении 8.

3.3.3 Описание принципов построения пользовательского интерфейса

В Android пользовательский интерфейс состоит из легко встраиваемых компонентов. Каждый компонент может быть создан разработчиком и может управлять отображением представления на экране. Компоненты должны быть описаны в layout.xml файле. Вся вёрстка происходит в xml. Программист может встраивать в layout файл как заранее определённые компоненты, так и написанные самостоятельно. В рамках дипломной работы, полностью новые View классы не были написаны.

View – базовый компонент для всех Android компонентов. Кроме того, есть еще ViewGroup, который является базовым для всех компонентов, обладающих возможностью хранить в себе другие компоненты.

В рамках дипломной работы, для большей части компонентов, базовым использовался ConstraintLayout. ConstraintLayout – достаточно новый вид layout, который создан для уменьшения количества иерархий layout, что влияет на производительность. ConstraintLayout позволяет располагать View друг относительно друга с помощью Constrains правил.

Для формирования списков в Android используются такие компоненты как ListView и RecyclerView. Их различие заключается в том, что RecyclerView переиспользует View, которые вышли за границы экрана и не видимы пользователю, таким образом, экономя память и производительность устройства, поскольку даже для бесконечного списка, системой будет создано только то количество View, которое помещается на экран. ListView подходит для формирования небольших списков.

Для конфигурации RecyclerView, ему необходимо передать layoutManager и adapter.

LayoutManager – класс ответственный за отображение элементов RecyclerView, за их пролистывание и размещение на экране

Adapter – класс–реализация паттерна проектирования Адаптер, является конвертером между данными и View. К основным функциям Adapter’а относится onCreateViewHolder – создающая View для RecyclerView, onBindViewHolder – производит установку значений в созданную View. Для создания новых View их необходимо создать из xml разметки используя LayoutInflater#inflate.

Пример конфигурации Adapter для RecyclerView компонента можно найти в приложении 9.

3.3.4 Описание использованных сторонних библиотек

Основные сторонние библиотеки, используемые в приложении:

- Kotlin – стандартная библиотека Kotlin-функций;
- Kotlin-coroutines – поддержка Kotlin-корутин;
- Android Material – библиотека компонентов в MaterialDesign стиле;
- Navigation – для облегчения навигации между фрагментами приложения. Помогает организовать удобную навигацию в Single Activity приложении;
- Koin – DI фремворк с поддержкой viewModel, scope и Kotlin DSL;
- OkHttp – библиотека инкапсулирующая логику сетевых запросов;
- Room – ORM для SqlLite.

3.3.5 Сборка и структура проекта

Для разработки приложения использовался gradle – система автоматической сборки, построенная на принципах ApacheAnt и ApacheMaven, но предоставляющая DSL на языках Groovy и Kotlin. Был разработан для расширяемых многопроектных сборок и поддерживает инкрементальные сборки, определяя какие компоненты дерева сборки не изменились и какие задачи, зависимые от этих частей, не требуют перезапуска.

Пример структуры android-приложения представлен в листинге 5.

```
.
|-- app
|   |-- src
|   |   |-- androidTest
|   |   |-- main
|   |   |   |-- java
|   |   |   |-- res
|   |   |   `-- AndroidManifest.xml
|   |   `-- test
|   |-- build.gradle
|   `-- proguard-rules.pro
|-- build.gradle
|-- settings.gradle
|-- gradle.properties
|-- gradlew
`-- gradlew.bat
```

Листинг 5 – Пример структуры Android приложения

Исходники приложения, располагаются в директории src. В папке res располагаются все ресурсы проекта (строки, переводы, layoust, anim, drawable, navigation).

Файл proguard-rules.pro содержит конфигурацию обфускации кода.

3.4 Вывод по главе 3

В данной главе рассмотрены основные архитектурные решения реализации проекта, описаны основные классы, которые использовались при написании кода. Также, разобрана система сборки каждого из проектов.

ЗАКЛЮЧЕНИЕ

В дипломной работе разработан ряд приложений. Одни приложения предназначены для использования интернет-магазина потенциальным клиентом магазина. Другие - для управления менеджерами интернет-магазином. Разработанные приложения позволяют клиентам осуществлять просмотр интересующих товаров, оформлять заказ и связываться с менеджерами сайта с помощью формы обратной связи. Менеджеры имеют возможность редактировать все доступные позиции, управлять их наличием/отображением на сайте, редактировать данные пользователей, предоставлять им специальные скидки. Система приложений обладает выделенным сервером на котором хранится база данных.

Для достижения цели дипломной работы были решены следующие задачи.

1. Проанализирована предметная область.
2. Спроектирована как общая архитектура ряда приложений, их способ сообщения, так и архитектура каждого приложения отдельно.
3. Выбраны средства разработки и приведено обоснование данного выбора.
4. Разработаны БД, API для связи сервера и клиентов, UI клинтских приложений.
5. Разработан REST-сервер, а также Web и Android приложения-клиенты.
6. Проанализирован и разработан пользовательский интерфейс с учетом основных тенденций и принципов, повышающих его удобство и позволяющий полностью реализовать необходимый функционал.

Предлагаемая разработка является актуальной, так как решения, представленные на рынке, обладают недостаточным функционалом. Разработанное приложение будет актуальным для людей и компаний

заинтересованных создании своего интернет-магазина оптовой торговли.

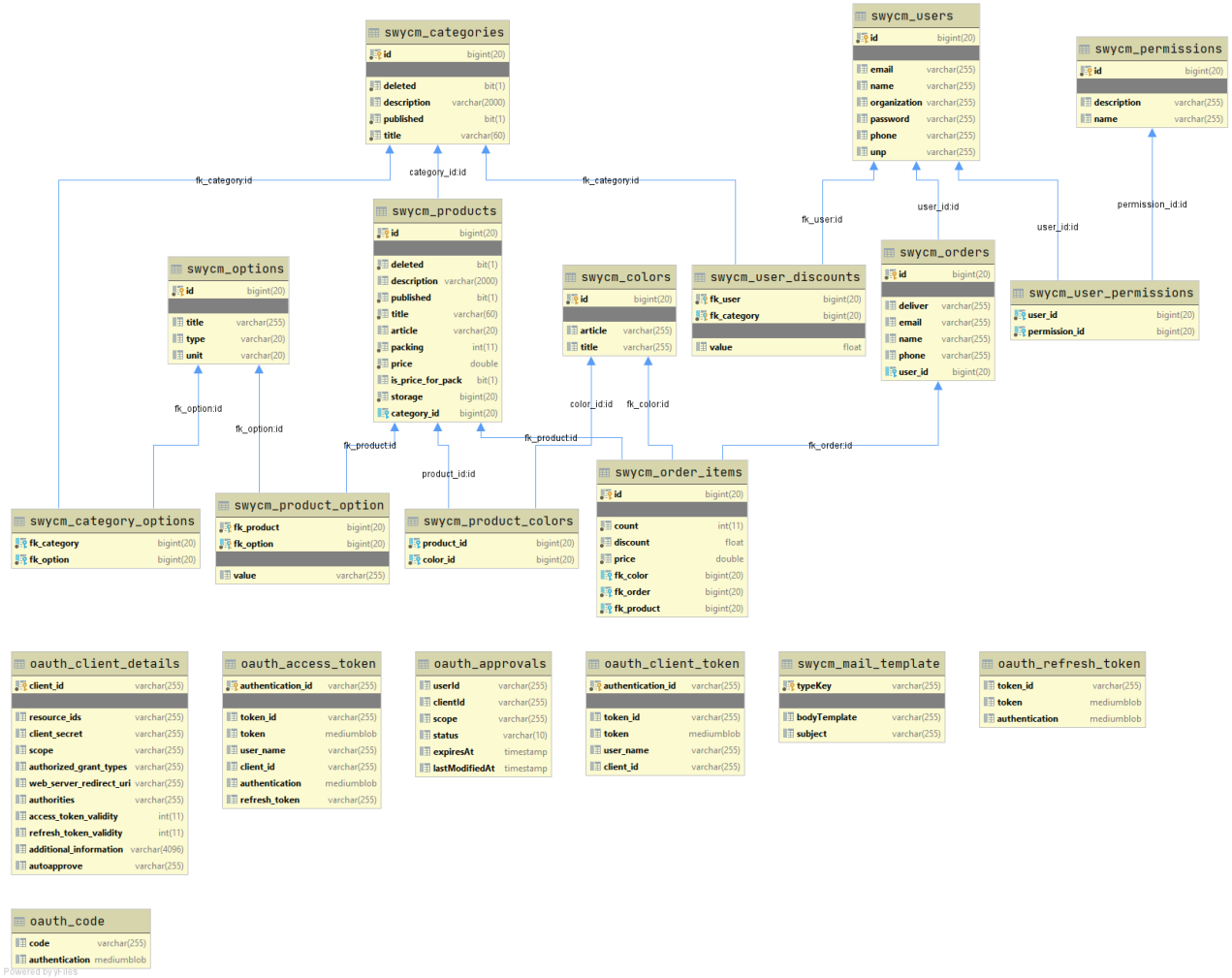
По дипломной работе имеется статья, материалы которой опубликованы в сборниках конференции в печатном и электронном виде, также в сборниках Гродненского Государственного университета «Наука - 2019».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Интерфес прикладного программирования [Электронный ресурс] // Stud ref. Режим доступа: https://bit.ly/2WVufwG - Дата доступа: 24.05.2020
2. Java Virtual Machine [Электронный ресурс] // Википедия. Свободная энциклопедия. Режим доступа: https://bit.ly/3ei6yoq - Дата доступа: 24.05.2020
3. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения. / Р. Мартин СПб.:Питер, 2018 – С. 85.
4. Собеседоваие по Java EE — Java Persistence API [Электронный ресурс] // Java study. Режим доступа: https://bit.ly/3cXySvO - Дата доступа: 24.05.2020
5. Протоколы электронной почты [Электронный ресурс] // Учебно-методические материалы для студентов кафедры АСОУИ. Режим доступа: https://bit.ly/3c0sQtc - Дата доступа: 24.05.2020
6. HTTP [Электронный ресурс] // MDN web docs. Режим доступа: https://mzl.la/2WWAnos - Дата доступа: 24.05.2020
7. Понятие клиент-серверных систем [Электронный ресурс] // StudFiles. Режим доступа: https://bit.ly/3d0eoCK - Дата доступа: 24.05.2020
8. Dmitry Jemerov. Kotlin in Action / Dmitry Jemerov, Svetlana Isakova 2016 – C. 6.
9. Выбор и описание программных средств и среды разработки реализации сайта [Электронный ресурс] // StudBooks. Режим доступа: https://bit.ly/2XtSdOE - Дата доступа: 24.05.2020
10. AngularJS [Электронный ресурс] // Википедия. Свободная энциклопедия. Режим доступа: https://bit.ly/36suORU - Дата доступа: 24.05.2020
11. Typescript [Электронный ресурс] // Википедия. Свободная энциклопедия. Режим доступа: https://bit.ly/36rr5ns - Дата доступа: 24.05.2020

ПРИЛОЖЕНИЕ 1

Схема БД



ПРИЛОЖЕНИЕ 2

Листинг AuthorizationService веб-клиента

```
import {Injectable} from '@angular/core';
import {
  HttpClient,
  HttpErrorResponse,
  HttpEvent,
  HttpHandler,
  HttpInterceptor,
  HttpRequest
} from "@angular/common/http";
import {BehaviorSubject, Observable} from "rxjs";
import {LocalStorageService} from "angular-2-local-storage";
import {TokenResponse} from "../model/token-response";
import {catchError, filter, finalize, switchMap, take, tap} from "rxjs/operators";
import {RegistrationModel} from "../model/registration-model";
import {Api} from "../constants";

@Injectable({
  providedIn: 'root'
})
export class AuthorizationService implements HttpInterceptor {

  private REGISTR_URL = Api.REGISTER;
  private TOKEN_URL = Api.TOKEN;

  private ACCESS_TOKEN = "access_token";
  private REFRESH_TOKEN = "refresh_token";
  private EXPIRES = "expires";
  private GRANT_TYPE = "grant_type";
  private USERNAME = "username";
  private PASSWORD = "password";
  private CLIENT_ID = "client_id";
  private CLIENT_SECRET = "client_secret";
  private CLIENT_ID_VALUE = "web";
  private CLIENT_SECRET_VALUE = "secret";
  private SCOPE = "scope";
  private SCOPE_READ = "read";

  private isRefreshingToken: boolean = false;
  private tokenSubject: BehaviorSubject<string> = new BehaviorSubject<string>(null);

  constructor(
    private http: HttpClient,
    private localStorage: LocalStorageService,
  ) {
  }

  get bearerToken(): string {
    return this.localStorage.get(this.ACCESS_TOKEN);
  }

  get refreshToken(): string {
    return this.localStorage.get(this.REFRESH_TOKEN)
  }

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const bearerToken = this.bearerToken;
    let req = this.appendTokenToRequest(request, bearerToken);

    if (bearerToken) {
      req = this.appendTokenToRequest(request, bearerToken);
    } else {
      req = request;
    }

    return next.handle(req).pipe(
      catchError(err => {
        if (err instanceof HttpErrorResponse) {
          switch ((<HttpErrorResponse>err).status) {
            case 401:
              return this.handle401Error(request, next);
            default: {
```

```
72                throw  err;
73              }
74            }
75          } else {
76            throw err;
77          }
78        })
79    )
80  }
81
82  login(username: string, password: string): Observable<TokenResponse> {
83    return this.getToken(username, password);
84  }
85
86  logout() {
87    this.cleanTokenData();
88  }
89
90  isAuthenticated(): boolean {
91    if (this.bearerToken) {
92      return true;
93    } else {
94      return false;
95    }
96  }
97
98  register(registrationModel: RegistrationModel) {
99    return this.http.post(this.REGISTR_URL, registrationModel)
100 }
101
102 private handle401Error(request: HttpRequest<any>, next: HttpHandler): Observable<any> {
103   if (!this.isRefreshingToken && this.refreshToken) {
104     this.isRefreshingToken = true;
105     this.tokenSubject.next(null);
106
107     return this.refreshAccessToken().pipe(
108       switchMap((token: TokenResponse) => {
109         if (token) {
110           this.tokenSubject.next(token.access_token);
111           this.saveTokenData(token);
112
113           return next.handle(this.appendTokenToRequest(request, token.access_token));
114         }
115       }),
116       catchError(() => {
117         this.cleanTokenData();
118         return next.handle(request);
119       }),
120       finalize(() => {
121         this.isRefreshingToken = false;
122       })
123     )
124   } else {
125     this.isRefreshingToken = false;
126
127     return this.tokenSubject
128       .pipe(filter(token => token != null),
129         take(1),
130         switchMap(token => {
131           return next.handle(this.appendTokenToRequest(request, token))
132         })
133       )
134   }
135 }
136
137 private getToken(username: string, password: string): Observable<TokenResponse> {
138   const url = this.TOKEN_URL;
139   const body = new FormData();
140
141   this.appendClientAuthData(body);
142   body.append(this.GRANT_TYPE, this.PASSWORD);
143   body.append(this.SCOPE, this.SCOPE_READ);
144   body.append(this.USERNAME, username);
145   body.append(this.PASSWORD, password);
146
147   return this.http.post<TokenResponse>(url, body)
148     .pipe(
149       tap(success => {
150         console.log("success" + success.access_token);
151         this.saveTokenData(success);
152       })
```

```
153       );
154     }
155
156     private refreshAccessToken(): Observable<TokenResponse> {
157       const refreshToken: string = this.refreshToken;
158       const url = this.TOKEN_URL;
159       const body = new FormData();
160
161       this.appendClientAuthData(body);
162       body.append(this.GRANT_TYPE, this.REFRESH_TOKEN);
163       body.append(this.REFRESH_TOKEN, refreshToken);
164
165       return this.http.post<TokenResponse>(url, body)
166     }
167
168     private saveTokenData(tokenData: TokenResponse) {
169       const currentTime = Date.now();
170       const expires = currentTime + (tokenData.expires_in * 1000);
171
172       this.localStorage.set(this.ACCESS_TOKEN, tokenData.access_token);
173       this.localStorage.set(this.REFRESH_TOKEN, tokenData.refresh_token);
174       this.localStorage.set(this.EXPIRES, expires);
175     }
176
177     private cleanTokenData() {
178       this.localStorage.remove(
179         this.ACCESS_TOKEN,
180         this.REFRESH_TOKEN,
181         this.EXPIRES
182       )
183     }
184
185     private appendClientAuthData(data: FormData) {
186       data.append(this.CLIENT_ID, this.CLIENT_ID_VALUE);
187       data.append(this.CLIENT_SECRET, this.CLIENT_SECRET_VALUE);
188     }
189
190     private appendTokenToRequest(request: HttpRequest<any>, token: string) {
191       return request.clone({
192         setHeaders: {
193           Authorization: 'Bearer ${token}'
194         }
195       });
196     }
197   }
```

ПРИЛОЖЕНИЕ 3

Листинг MgrProductService веб-клиента

```
import {Injectable} from '@angular/core';
import {HttpClient, HttpParams} from "@angular/common/http";
import {Api} from "../../constants";
import {Product} from "../../model/product";
import {Observable} from "rxjs";
import {ProductDetails} from "../../model/product-details";

@Injectable({
  providedIn: 'root'
})
export class MgrProductService {

  private BASE_URL = Api.MGR_PRODUCTS;
  private DELETED_PATH = "/deleted";
  private PUBLISHED_PATH = "/published";

  constructor(
  private http: HttpClient
  ) { }

  getByCategoryId(id: number): Observable<Product[]> {
    const url = this.BASE_URL;
    const options = {
      params: new HttpParams().set('categoryId', id.toString())
    };

    return this.http.get<Product[]>(url, options)
  }

  getDetailsById(id: number): Observable<ProductDetails> {
    const url = this.BASE_URL + "/" + id;

    return this.http.get<ProductDetails>(url)
  }

  toggleDeleted(id: number): Observable<boolean> {
    const url = this.BASE_URL + this.DELETED_PATH;

    return this.http.put<boolean>(url, id)
  }

  togglePublihsed(id: number) {
    const url = this.BASE_URL + this.PUBLISHED_PATH;

    return this.http.put<boolean>(url, id)
  }

  update(productDetails: ProductDetails): Observable<ProductDetails> {
    const url = this.BASE_URL;

    return this.http.put<ProductDetails>(url, productDetails);
  }
}
```

ПРИЛОЖЕНИЕ 4

Листинг ProductAdd компонента веб-клиента

```
<mat-horizontal-stepper [linear]="true"
                        (selectionChange)="onSelectionChange($event)">
    <mat-step label="User primary data"
              [stepControl]="primaryDataForm">
        <form [formGroup]="primaryDataForm" class="container">
            <mat-form-field>
                <input matInput
                       placeholder="Unp"
                       formControlName="unp">
                <mat-error *ngIf="!isValidFormField('unp')">
                    Unp must contain exactly 9 numbers
                </mat-error>
            </mat-form-field>

            <mat-form-field>
                <input matInput
                       placeholder="email"
                       formControlName="email">
            </mat-form-field>

            <mat-form-field>
                <input matInput
                       placeholder="phone"
                       formControlName="phone">
                <span рубmatSuffix>.</span>
                <mat-error *ngIf="!isValidFormField('phone')">
                    Enter valid float value from 0.0 to 9999.0
                </mat-error>
            </mat-form-field>

            <mat-form-field>
                <input matInput
                       placeholder="name"
                       formControlName="name">
                <span штmatSuffix>.</span>
                <mat-error *ngIf="!isValidFormField('name')">
                    Enter valid int value from 1 to 9999
                </mat-error>
            </mat-form-field>

            <mat-form-field>
                <input matInput
                       placeholder="name"
                       formControlName="name">
                <span штmatSuffix>.</span>
                <mat-error *ngIf="!isValidFormField('name')">
                    Enter valid int value from 1 to 9999
                </mat-error>
            </mat-form-field>
            <button color="accent"
                    mat-raised-button
                    matStepperNext
                    [disabled]="primaryDataForm.invalid">
                Next
            </button>

        </form>
    </mat-step>

    <mat-step label="Product colors"
              [stepControl]="colorsDataForm">
        <form [formGroup]="colorsDataForm" class="container">
            <div class="two_cols">
                <mat-checkbox *ngFor="let color of colorsArray; let i = index" [formControlName]="i"
                    >
                    {{color.article}} — {{color.title}}
                </mat-checkbox>
            </div>
            <div>
                <button color="accent" mat-raised-button matStepperPrevious>Back</button>
                <button color="accent" mat-raised-button matStepperNext>Next</button>
            </div>
        </form>
```

```
73      </mat-step>
74
75      <mat-step label="Product options data"
76                [stepControl]="optionsDataForm">
77          <form [formGroup]="optionsDataForm" class="container">
78              <ng-container *ngFor="let option of optionArray; let i = index;"
79                            [ngSwitch]="option.type">
80                  <mat-slide-toggle *ngSwitchCase="OptionType.BOOL"
81                                    [formControlName]="i">
82                      {{option.title}}
83                  </mat-slide-toggle>
84
85                  <mat-form-field *ngSwitchCase="OptionType.INT">
86                      <input matInput
87                             formControlName="{{i}}"
88                             [placeholder]="option.title">
89                      <span *ngIf="option.unit" matSuffix>{{option.unit}}</span>
90                  </mat-form-field>
91
92                  <mat-form-field *ngSwitchCase="OptionType.FLOAT">
93                      <input matInput
94                             formControlName="{{i}}"
95                             [placeholder]="option.title">
96                      <span *ngIf="option.unit" matSuffix>{{option.unit}}</span>
97                  </mat-form-field>
98
99                  <mat-form-field *ngSwitchDefault>
100                     <input matInput
101                            formControlName="{{i}}"
102                            [placeholder]="option.title">
103                     <span *ngIf="option.unit" matSuffix>{{option.unit}}</span>
104                 </mat-form-field>
105             </ng-container>
106
107             <div>
108                 <button color="accent" mat-raised-button matStepperPrevious>Back</button>
109                 <button color="accent" mat-raised-button matStepperNext
110                         [disabled]="optionsDataForm.invalid">Next
111                 </button>
112             </div>
113         </form>
114     </mat-step>
115
116     <mat-step label="Check and submit">
117         <div class="container">
118             <mat-card>
119                 <mat-card-title>{{checkPrimaryLine}}</mat-card-title>
120                 <mat-card-subtitle>{{checkSecondaryLine}}</mat-card-subtitle>
121                 <mat-card-content>
122                     <mat-divider></mat-divider>
123
124                     <table class="check_table">
125                         <tr>
126                             <td>In package:</td>
127                             <td>{{productModel.packing | count}}</td>
128                         </tr>
129                         <tr>
130                             <td>Price:</td>
131                             <td>{{productModel.price | price}}</td>
132                         </tr>
133                         <tr>
134                             <td>Is price for pack:</td>
135                             <td>{{productModel.priceForPack}}</td>
136                         </tr>
137                         <tr>
138                             <td>Package price:</td>
139                             <td>{{productModel | price}}</td>
140                         </tr>
141                     </table>
142
143                     <mat-divider></mat-divider>
144
145                     <table class="check_table">
146                         <ng-container *ngFor="let option of productModel.options">
147                             <tr>
148                                 <td>{{option.title}}</td>
149                                 <td>{{option.value}} {{option.unit}}</td>
150                             </tr>
151                         </ng-container>
152                     </table>
```

```
153
154                     <mat-divider></mat-divider>
155
156                     <mat-list *ngIf="productModel.colors && productModel.colors.length > 0">
157                         <mat-list-item *ngFor="let color of productModel.colors">
158                             {{color.article}} {{color.title}}
159                         </mat-list-item>
160                     </mat-list>
161
162                 </mat-card-content>
163             </mat-card>
164
165             <div>
166                 <button color="accent" mat-raised-button matStepperPrevious>Back</button>
167                 <button color="accent" mat-raised-button matStepperNext
168                         (click)="onStepComplete(StepType.CHECK)"
169                         [disabled]="(primaryDataForm.invalid || optionsDataForm.invalid)">Submit
170                 </button>
171             </div>
172         </div>
173     </mat-step>
174 </mat-horizontal-stepper>
```

ПРИЛОЖЕНИЕ 5

Листинг CSS для ProductAdd компонента веб-клиента

```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;
  grid-auto-rows: minmax(55px, auto);
  align-items: center;
}

.container > * {
  grid-column-start: 2;
  grid-column-end: 3;
  vertical-align: center;
}

.container > .two_cols {
  display: grid;
  grid-template-columns: repeat(2, 1fr);
  grid-column-gap: 20px;
  grid-auto-rows: minmax(40px, auto);
}

.container > :last-child {
  display: grid;
  grid-template-columns: repeat(2, 1fr);
  grid-column-gap: 20px;
}

button {
  width: 100%;
}

.check_table {
  padding: 10px 0;
}

.check_table tr {
  padding: 0;
  line-height: 1.5;
}

.check_table tr td:first-child {
  padding-right: 25px;
}

.mat-horizontal-stepper-header {
  pointer-events: none !important;
}

@media only screen and (max-width: 700px) {
  .container {
    grid-template-columns: 1fr;
  }

  .container * {
    grid-column-start: 1;
    grid-column-end: 2;
  }
}
```

ПРИЛОЖЕНИЕ 6

Листинг AuthTokenInterceptor Android-клиента

```
1  package com.github.swalffy.magnat_manager.utils.networking.interceptor
2
3  import com.github.swalffy.magnat_manager.features.login.RefreshTokenUsecase
4  import com.github.swalffy.magnat_manager.utils.*
5  import com.github.swalffy.magnat_manager.utils.common.SharedPrefs
6  import okhttp3.Interceptor
7  import okhttp3.Response
8  import org.koin.core.KoinComponent
9  import org.koin.core.inject
10
11 class AuthTokenInterceptor(
12     private val refreshTokenUsecase: RefreshTokenUsecase
13 ) : Interceptor, KoinComponent {
14
15     private val preferences: SharedPrefs by inject()
16
17     override fun intercept(chain: Interceptor.Chain): Response {
18         val newRequest = chain.request().newBuilder().apply {
19             preferences.bearerToken?.let {
20                 addHeader("Authorization", "Bearer $it")
21             }
22         }.build()
23
24         var response = chain.proceed(newRequest)
25         if (response.code() == 401) {
26             preferences.bearerToken = null
27             val refreshToken = preferences.refreshToken
28
29             if (refreshToken?.isNotEmpty() == true) {
30                 when (tryRestoreSession(refreshToken)) {
31                     is Success -> {
32                         newRequest.newBuilder().run {
33                             preferences.bearerToken?.let {
34                                 addHeader("Authorization", "Bearer $it")
35                             }
36                             build()
37                         }.let { response = chain.proceed(it) }
38                     }
39                     is Error -> {
40                         preferences.refreshToken = null
41                         preferences.bearerToken = null
42                     }
43                 }
44             }
45         }
46         return response
47     }
48
49     private fun tryRestoreSession(refreshToken: String) =
50         refreshTokenUsecase.performTokenRefresh(refreshToken)
51 }
```

ПРИЛОЖЕНИЕ 7

Листинг ProductListGetAllUseCase Android-клиента

```
package com.github.swalffy.magnat_manager.features.products

import com.github.swalffy.magnat_manager.utils.Error
import com.github.swalffy.magnat_manager.utils.Result
import com.github.swalffy.magnat_manager.utils.Success
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.withContext

class ProductListGetAllUseCase(
    val productsRepository: ProductListRepository
) {

    suspend fun loadProducts(categoryId: Long): Result<List<ProductModel>> {
        return coroutineScope {
            val products = withContext(Dispatchers.IO) {
                productsRepository.getAllProducts(categoryId = categoryId)
            }

            if (products?.isNotEmpty() == true) {
                products.map {
                    ProductModel(
                        id = it.id,
                        title = it.title,
                        article = it.article,
                        deleted = it.deleted,
                        published = it.published,
                        price = if (it.priceForPack)
                            it.price * it.packing
                        else
                            it.price
                    )
                }.let { Success(it) }
            } else {
                Error(RuntimeException("No products"))
            }
        }
    }
}
```

ПРИЛОЖЕНИЕ 8

Листинг HttpRequestManager Android-клиента

```
package com.github.swalffy.magnat_manager.utils.networking.core

import android.util.Log
import com.github.swalffy.magnat_manager.utils.common.hashOf
import com.github.swalffy.magnat_manager.utils.database.entity.network.NetworkRequestCacheDao
import com.github.swalffy.magnat_manager.utils.database.entity.network.NetworkRequestCacheRecord
import com.github.swalffy.magnat_manager.utils.networking.core.RequestMethod.BodyType
import com.github.swalffy.magnat_manager.utils.networking.interceptor.AuthTokenInterceptor
import com.github.swalffy.magnat_manager.utils.networking.model.FormDataModel
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.async
import kotlinx.coroutines.withContext
import okhttp3.*
import okhttp3.HttpUrl.Companion.toHttpUrlOrNull
import okhttp3.MediaType.Companion.toMediaTypeOrNull
import okhttp3.RequestBody.Companion.toRequestBody
import java.io.InputStream
import java.util.concurrent.TimeUnit

private const val HTTP_CALL_TAG = "sw_HTTP_CALL"

class HttpRequestManager(
    private val requestCacheDao: NetworkRequestCacheDao,
    private val networkConfig: NetworkClientConfig,
    private val jsonConverter: JsonConverter
) {

    private val client: OkHttpClient = OkHttpClient.Builder()
        .addInterceptor(AuthTokenInterceptor())
        .callTimeout(networkConfig.timeoutMillis, TimeUnit.MILLISECONDS)
        .build()

    suspend fun <T> request(
        method: RequestMethod,
        path: String,
        queryParams: Map<String, Any?>? = null,
        cacheControl: CacheControl<T>? = null,
        onSuccess: ((InputStream) -> T)? = null,
        onError: ((Int, InputStream?) -> T?)? = { code, _ ->
            Log.w(HTTP_CALL_TAG, "HttpRequestManager request: some error in call $method $path with $code")

            null
        },
        onException: ((Throwable) -> T)? = { throw it }
    ): T? = withContext(Dispatchers.IO) {
        val requestStartTime = System.currentTimeMillis()

        val clearExpiredJob = async { clearExpiredRequests(requestStartTime) }

        val request = buildRequest(
            method = method,
            path = path,
            queryParams = queryParams
        )

        if (cacheControl?.isForceRequest == false) {
            clearExpiredJob.await()

            val cachedRecord = requestCacheDao.getByRequestHash(hashOf(method, request.url.toString()))

            if (cachedRecord != null && cachedRecord.expires > requestStartTime) {
                Log.d(HTTP_CALL_TAG, "HttpRequestManager request: $method $path already cached(expire in ${cachedRecord.expires - cacheControl.expiration})")
                return@withContext null
            }
        }

        runCatching {
            client.newCall(request)
```

```
                .execute().use { response ->
                    Log.d(HTTP_CALL_TAG, "HttpRequestManager request: $method $path complete with ${
                        response.code}")

                    if (response.isSuccessful) {
                        cacheControl?.let { cache ->
                            NetworkRequestCacheRecord(
                                id = hashOf(method.toString(), request.url),
                                url = request.url.toString(),
                                method = method.toString(),
                                paramsHash = method.hashCode(),
                                expires = requestStartTime + cache.expiration
                            ).let(requestCacheDao::insert)
                        }

                        response.body?.byteStream()
                            ?.let { stream ->
                                onSuccess?.invoke(stream)
                                    ?.also { cacheControl?.onCache?.invoke(it) }
                            } ?: onError?.invoke(-1, null)
                    } else {
                        onError?.invoke(response.code, response.body?.byteStream())
                    }
                }
        }.getOrElse {
            Log.w(HTTP_CALL_TAG, "HttpRequestManager request: ", it)
            onException?.invoke(it)
        }
    }

    private fun buildRequest(
        method: RequestMethod,
        path: String,
        queryParams: Map<String, Any?>? = null
    ): Request {
        val url = networkConfig.host.toHttpUrlOrNull()
            ?.newBuilder()
            ?.addPathSegments(path)
            ?.also {
                queryParams?.forEach { (key, value) ->
                    it.addEncodedQueryParameter(key, value.toString())
                }
            }?.build()
            ?: error("Can't build url. Base[${networkConfig.host}] path: [$path]")

        val body = (method as? RequestMethod.RequestWithBody<*>)
            ?.let { requestWithBody ->
                when (requestWithBody.bodyType) {
                    BodyType.JSON -> jsonConverter.toJson(requestWithBody.body)
                        .toRequestBody("application/json; charset=utf-8".toMediaTypeOrNull())

                    BodyType.FORM_DATA -> (requestWithBody.body as? FormDataModel)
                        ?.asMap
                        ?.let { bodyMap ->
                            MultipartBody.Builder()
                                .setType(MultipartBody.FORM)
                                .also {
                                    bodyMap.forEach { (key, value) -> it.addFormDataPart(key, value)
                                        }
                                }.build()
                        } ?: throw error("Form data body should be Map<String, String>")
                }
            }

        return Request.Builder()
            .method(method.toString(), body)
            .url(url)
            .build()
    }

    private suspend fun clearExpiredRequests(currentTime: Long) {
        requestCacheDao.dropExpired(currentTime)
    }
}
```

ПРИЛОЖЕНИЕ 9

Листинг ProductListRecyclerAdapter Android-клиента

```
1  package com.github.swalffy.magnat_manager.features.products
2
3  import android.view.View.*
4  import android.view.ViewGroup
5  import androidx.recyclerview.widget.RecyclerView
6  import kotlinx.android.extensions.LayoutContainer
7  import kotlinx.android.synthetic.main.list_item_product.*
8
9  class ProductListRecyclerAdapter(
10     private val onClick: (Long) -> Unit
11 ) : RecyclerView.Adapter<ProductListRecyclerAdapter.ProductListItemHolder>() {
12
13     private val items = mutableListOf<ProductModel>()
14
15     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ProductListItemHolder {
16         val inflatedView = parent.inflate(R.layout.list_item_product, false)
17         return ProductListItemHolder(inflatedView)
18     }
19
20     override fun getItemCount(): Int = items.size
21
22     override fun onBindViewHolder(holder: ProductListItemHolder, position: Int) {
23         val product = items[holder.adapterPosition]
24         holder.bind(product)
25     }
26
27     fun cleanAddAll(newItems: List<ProductModel>) {
28         items.clear()
29         items.addAll(newItems)
30         notifyDataSetChanged()
31     }
32
33     inner class ProductListItemHolder(
34         override val containerView: View
35     ) : RecyclerView.ViewHolder(containerView), LayoutContainer {
36
37         init {
38             containerView.setOnClickListener {
39                 val productItem = items[adapterPosition]
40
41                 onClick.invoke(productItem.id)
42             }
43         }
44
45         fun bind(product: ProductModel) {
46             text_title.text = product.title
47             text_article.text = product.article
48             view_deleted.visibility = if (product.deleted) VISIBLE else INVISIBLE
49             view_published.visibility = if (product.published) INVISIBLE else VISIBLE
50             text_price.text = product.price.toString()
51         }
52     }
53 }
```