МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»
Профиль подготовки: Вычислительные машины, комплексы, системы и сети

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

на тему

Мобильное приложение для подбора одежды с учётом погодных условий

Студент 03.06.20 Чиркин Кирилл Денисович
(подпись, дата) (ФИО полностью)

Руководитель 03.06.20 Митрохин М.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр Кучин А.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 03.06.2020 № 12)

Заведующий кафедрой М.А. Митрохин
(подпись) (фамилия, инициалы)

Работа защищена с отметкой Отлично (протокол заседания ГЭК от 11 июня 20 г. № 5)

Секретарь ГЭК Е.И. Гурин
(подпись) (фамилия, инициалы)

Пенза 2020

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____М.А. Митрохин
«30» апреля 2020г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

1. Студент Чиркин Кирилл Денисович гр.16ВВ1 факультета вычислительной техники.

2. Тема работы: «Мобильное приложение для подбора одежды с учётом погодных условий»

Тема утверждена приказом ПГУ №0335а/о от "30" апреля 2020 г.

3. Руководитель выпускной квалификационной работы бакалавра д.т.н, доцент каф. ВТ ПГУ Митрохин М.А.

4. Задание на работу (назначение разработки, исходные данные и т.п.).

Назначение разработки – мобильное клиент-серверное приложение для ОС Android, предназначенное для подбора одежды с учетом погодных условий. Необходимо учитывать гардероб пользователя, разработать функционал для совершенствования алгоритма подбора одежды и реализовать клиент-серверное взаимодействие.

5. Перечень подлежащих разработке вопросов.

Требуется провести анализ предметной области, выбрать архитектуру приложения, определить языки программирования для клиента и сервера, выбрать способ взаимодействия,

разработать протокол взаимодействия, реализовать клиент и сервер.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Анализ предметной области	5	27.05.20	
2.	Выбор архитектуры приложения, языков программирования, способа взаимодействия	5	28.05.20	
3.	Разработка протокола взаимодействия	15	29.05.20	
4.	Разработка серверной части	20	30.05.20	
5.	Разработка клиентской части	20	31.05.20	
6.	Тестирование и отладка	15	02.06.20	
7.	Оформление выпускной работы	20	03.06.20	

Дата выдачи задания "30" апреля 2020 г.

Руководитель выпускной квалификационной работы бакалавра

(подпись, дата)

Задание к исполнению принял студент

(подпись, дата)

Работу к защите допустить

Декан факультета

(подпись, дата)

Реферат

Бакалаврская работа содержит 4 листа, 32 рисунка, 1 таблицу, 4 источника, 3 приложения.

МОБИЛЬНОЕ ПРИЛОЖЕНИЕ, КЛИЕНТ-СЕРВЕРНАЯ СИСТЕМА, ANDROID, MVVM, РЕКОМЕНДАТЕЛЬНАЯ СИСТЕМА, ПОГОДА, ОДЕЖДА

Цель работы — разработать мобильное приложение для подбора одежды с учётом погодных условий.

В результате проделанной работы было разработано приложение, обеспечивающее выполнение следующих функций: отображение текущей погоды и прогноза погоды, авторизация пользователя, возможность составления собственного гардероба, смены пола пользователя, получения уведомлений о посещении приложения, адаптация алгоритма подбора одежды для конкретного пользователя по признаку индивидуального восприятия температуры.

В дальнейшем приложение можно доработать, например, добавив функционал подбора одежды по стилям или на несколько дней вперед.

СОДЕРЖАНИЕ

Реферат....4

Введение....7

1 Анализ предметной области и имеющихся аналогов....9

1.1 Постановка задачи....9

1.2 Актуальность....10

1.3 Обзор аналогов....10

1.4 Выбор средств разработки клиентской части приложения....11

1.4.1 Выбор языка программирования....11

1.4.2 Выбор среды разработки....12

1.4.3 Описание используемых инструментов и библиотек....13

1.5 Выбор средств разработки серверной части приложения....13

1.5.1 Выбор языка программирования....13

1.5.2 Выбор среды разработки....14

1.5.3 Описание используемых инструментов и библиотек....14

1.5.4 Выбор СУБД....15

1.6 Механизмы взаимодействия со сторонними сервисами....15

1.6.1 API погоды....15

2 Разработка структуры приложения....17

2.1 Описание архитектуры программы....17

2.2 Описание архитектуры базы данных....26

3 Программная реализация....30

3.1 Реализация клиента....30

3.1.1 Описание работы экрана AuthActivity....30

3.1.2 Описание работы экрана MainActivity....30

3.2 Реализация сервера.....33
3.2.1 Описание работы MainService.....33
3.2.2 Описание работы WeatherService.....34
3.2.3 Описание работы WardrobeService.....35
3.2.4 Описание работы UserService.....35
3.2.5 Описание работы WhatToWearService.....36
3.3 Руководство пользователя.....37
4 Тестирование программы.....42
4.1 Тест-кейсы.....42
4.2 Результаты тестирования.....43
5 Безопасные условия труда оператора ЭВМ.....48
5.1 Аспекты производственной безопасности при работе на ПК.....48
5.1.1 Психофизиологические факторы.....48
5.1.2 Защита от излучений.....49
5.1.3 Оборудование рабочих мест с ПК.....49
5.1.4 Электробезопасность.....49
5.1.5 Освещение рабочего места.....50
5.2 Пожарная безопасность.....51
Заключение.....53
Список используемых источников.....54
Приложение А.....55
Приложение В.....83
Приложение С.....99

ВВЕДЕНИЕ

С каждым днем программное обеспечение играет в жизни людей все более заметную роль, давая нам возможность получать информацию, принимать решения "в режиме реального времени" и вообще вести более продуктивную и интересную жизнь. Вначале, когда количество доступных компьютеров было весьма невелико, каждый из них использовался в режиме коллективного доступа. Сегодня триумфальное шествие закона Мура подарило нам более демократичный и распределенный вид вычислительной техники — настольные и переносные (лэптопы, ноутбуки) "персональные компьютеры". В настоящее время мы являемся свидетелями еще более глубокого эволюционного процесса. Персональные компьютеры и централизованные серверы никуда не делись, но их постепенно начинают окружать мириады мобильных устройств, обеспечивающих дальнейшую децентрализацию вычислительных мощностей. Это расширяет сферу применения программных приложений на доселе неведомые области. Те самые люди, которые просматривают Web-страницы, обмениваются сообщениями электронной почты, проводят время за электронными играми, совершают электронные покупки или каким-то иным образом интерактивно взаимодействуют с миром сетевой информации при помощи настольных компьютеров, все чаще предпочитают захватывать с собой "в дорогу" ту или иную отдельную частичку этих возможностей.

В наши дни мобильные устройства предлагают разработчикам и конечным пользователям такие уникальные возможности для получения информации, обогащения знаниями и развлечения, которые еще несколько лет тому назад трудно было даже себе представить. Каждый, кто в течение более или менее длительного времени пользовался современными смартфонами или устройствами с возможностями Wi-Fi, мог сам убедиться в том, какие фантастические вещи сейчас становятся доступными. Ничто не отражает смысл выражения "информация на кончиках пальцев" так буквально, как

вычислительное устройство, которое можно в любой момент достать из кармана и сразу же начать работать с интересующей вас информацией.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ИМЕЮЩИХСЯ АНАЛОГОВ

1.1 Постановка задачи

Требуется разработать мобильное приложение для отображения набора одежды в зависимости от погодных условий. Мобильное приложение должно обеспечивать:

- Авторизация пользователя
- Получение подходящего набора одежды для текущей погоды и прогноза на день
- Просмотр погоды и прогноза погоды на текущий день
- Редактирование собственного гардероба для персонализированного подбора одежды
- Форму обратной связи для возможности предложить разработчикам добавить новые виды одежды в гардероб
- Настройку профиля пользователя

Созданное приложение должно поддерживаться на ОС Android версии 5.0 и выше.

Для этого приложения, соответственно, требуется написать серверную часть. Сервер должен обеспечивать:

- Авторизацию пользователя
- Изменения данных пользователя, такие как добавление вещей в гардероб или изменение настроек
- Выдачу категорий одежды и самих вещей
- Получение и выдачу погоды и прогноза погоды
- Доступ к статическим изображениям (одежды и человечка)
- Выдачу набора одежды для текущей погоды

1.2 Актуальность

Современный человек постоянно куда-то спешит. У него куча важных дел, повсюду отвлекающий информационный шум и острая нехватка времени. Люди стремятся быть более продуктивными. Чтобы помочь им в этом программисты стали разрабатывать множество приложений помощников, например, для учета финансовых расходов, работы с банковскими счетами, организации списка задач, просмотра карты, заказа такси, работы с государственными услугами («Госуслуги») и так далее. Даже существует программа, помогающая пить оптимальное количество воды для организма.

Имеется один важный аспект, который делает человека продуктивнее. Это комфортная погода. От нее зависит настроение и здоровье индивида. Если, например, жильца по пути на работу застанет резкое похолодание или неожиданный ливень, то весь день может быть испорчен (а если психологическое состояние человека плачевное, то последствия неподготовленности к такой погоде могут «подкосить» человека надолго).

Для решения вопроса подбора одежды в соответствии с погодой было принято решение сделать приложение «WannaDress» для платформы Android.

1.3 Обзор аналогов

На платформе Android существует множество конкурентов. Главным из них является «OshareWeather». Это приложение было установлено более 500 тысяч раз (что показывает востребованность такого типа программ), обладает большим набором предлагаемой одежды и неплохим алгоритмом подбора. Но этот продукт имеет существенные недостатки, а именно: ориентируется только на женскую аудиторию, преимущественно японскую, практически не поддерживается и имеет устаревший дизайн, имеет проблемы с локализацией. Остальные приложения, такие как «What 2 Wear?», «Swapparel» и т.д. обладают куда более меньшей аудиторией и сильно уступают в плане дизайна и проработки функционала, хотя иногда обладают интересными возможностями,

например, возможностью добавления фотографий своей одежды в гардероб или подбора одежды по принципу цветового сочетания.

На платформе iOS был замечен единственный достойный аналог – «Swackett», который создает впечатление законченного решения. К сожалению, не удалось узнать количество установок, возможно, магазин приложений App Store не предоставляет эти данные неавторизованным пользователям, но судя по степени проработанности можно предположить, что этот продукт обладает немалой аудиторией.

1.4 Выбор средств разработки клиентской части приложения

1.4.1 Выбор языка программирования

Для разработки под ОС Android изначально существовал только язык Java. Еще использовался (и используется) C++ для ресурсоемких операций. Со временем был добавлен язык Kotlin в качестве замены Java. Именно поэтому для реализации мобильного приложения был выбран Kotlin, но сначала рассмотрим вкратце язык Java, который послужил его основой.

Java — статически типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle).

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия

программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности.

Kotlin — это статически типизированный язык программирования, работающий поверх JVM и разрабатываемый компанией JetBrains.

Авторы ставили целью создать язык более лаконичный и типобезопасный, чем Java, и более простой, чем Scala. Следствием упрощения по сравнению со Scala стали также более быстрая компиляция и лучшая поддержка языка в IDE. Язык полностью совместим с Java, что позволяет java-разработчикам постепенно перейти к его использованию, а также дает доступ к использованию огромного количества библиотек, написанных для языка java. Главным преимуществом языка считается краткость, лаконичность, чистота и понятность кода, наличие синтаксического сахара.

На конференции для разработчиков Google I/O 2019 компания Google (известная как владелец ОС Android) объявила Kotlin **основным языком** для разработки Android приложений.

1.4.2 Выбор среды разработки

Для разработки мобильных приложений под платформу Android существует только одна среда разработки – Android Studio, которая и была выбрана. Если быть точнее, то есть альтернативный вариант в виде использования IDE IntellijIDEA с использованием плагина для разработки под Android. Но в итоге эта комбинация дает среду, которая практически идентична Android Studio, ведь последняя как раз и разработана на базе IntellijIDEA. Ранее, до появления Android Studio, разработчики пользовались средой Eclipse с установленным плагином для Android, но на данный момент этот способ сильно устарел, поэтому не рассматривается в качестве альтернативы.

1.4.3 Описание используемых инструментов и библиотек.

Для разработки мобильного приложения был использован стандартный набор инструментов - Android SDK (Softwate Development Kit). SDK содержит классы для работы с графическим интерфейсом, операционной системой, а также утилиты для сборки и отладки приложений.

Для загрузки изображений через интернет и для работы с ними использована библиотека Glide 4, которая предоставляет множество функций для работы с картинками, например обрезка или применение различных стратегий кеширования.

Для взаимодействия с сервером через http протокол использованы библиотеки OkHttp3 и Retrofit 2. Обмен данными происходит в формате json, поэтому для сериализации и десериализации подключена библиотека Gson.

В приложении работа с данными сделана с помощью реактивного программирования, поэтому использована библиотека RxJava2, а также ее различные расширения.

Для внедрения зависимостей в kotlin использована библиотека koin.

Отслеживание утечек памяти – библиотека LeakCanary.

1.5 Выбор средств разработки серверной части приложения

1.5.1 Выбор языка программирования

Для разработки серверной части приложения также был выбран язык Kotlin, который уже был рассмотрен в пункте 1.4.1.

Причины выбора Kotlin для разработки сервера:

- Строгая типизация и null-безопасность (null-safety - минимизация рисков получения ошибки NullPointerException, которая часто встречается в Java)
- Мультипарадигменность (возможность писать код в разных стилях, например, в ООП или функциональном)
- Краткий и лаконичный синтаксис

- Приемлемая производительность
- Большое количество библиотек, которые написаны как для Java, так и для Kotlin
- Этот язык используется на клиентской стороне, что может позволить иметь общую кодовую базу для бизнес-логики или моделей данных
- Возможность запуска программ на большинстве современных ОС при наличии JVM.
- Наличие опыта разработки и личное предпочтение

Существуют популярные альтернативы: Python, C#, PHP и т.д. У C# имеются сложности с запуском программ не на ОС Windows, а Python и PHP, по субъективному мнению, заметно уступают в плане синтаксиса и возможностей языка. У Kotlin имеется только один существенный недостаток – это очень сложные и перегруженные инструменты для сборки приложений, которые пришли «по наследству из корпоративного мира» языка Java.

1.5.2 Выбор среды разработки

Так как Kotlin и IDE IntellijIDEA разработаны одной компанией под названием JetBrains, то логично будет взять именно эту IDE, потому что в ней по умолчанию поставляется компилятор и официальная поддержка разработки на Kotlin. А еще Android Studio, используемая для разработки клиента, сделана на базе IntellijIDEA, поэтому при переключении между этими двумя средами интерфейс и набор горячих клавиш будет оставаться практически идентичным, что упростит процесс разработки.

1.5.3 Описание используемых инструментов и библиотек.

Для создания REST API был выбран фреймворк Ktor. Причиной послужила относительная простота освоения и официальная поддержка от JetBrains. Альтернативой мог бы послужить, например, известный фреймворк Spring Boot, но он все же больше подходит для крупных корпоративных веб-приложений, нежели для сервера мобильного приложения.

Для JSON сериализации и десериализации использована библиотека kotlinx.serialization. Причиной послужила не только поддержка от компании JetBrains, но и к тому же поддержка этой библиотеки для работы с фреймворком Ktor и простота использования.

Для тестирования сервера была использована утилита Postman. Эта программа позволяет с помощью удобного графического интерфейса выполнять любые виды HTTP запросов, сохранять часто используемые запросы и даже создавать собственные API в целях тестирования клиентского ПО.

1.5.4 Выбор СУБД

Для хранения основной информации, с которой будет работать сервер, была выбрана СУБД PostgreSQL. Причиной тому служат 3 основные вещи. Первая – открытый исходный код и, соответственно, возможность использовать базу данных полностью бесплатно. Вторая – эта СУБД является одной из самых активно развивающихся и используемых в индустрии, обладает продвинутым функционалом. И третья, самая простая, что PostgreSQL поддерживает язык запросов SQL и строгую схему данных.

Для сбора «сырых» данных, которые планируется использовать при обучении нейронной сети, была выбрана NoSQL база данных MongoDB. Главной причиной является самая большая популярность среди NoSQL решений. И с точки зрения функционала эта СУБД подходит, так как у нее нет строгой схемы (что характерно для большинства NoSQL решений) и можно сохранять массивы данных с большой скоростью без какой-либо дополнительной обработки. Подробнее смысл наличия второй базы данных будет рассмотрен в пункте 2.1.

1.6 Механизмы взаимодействия со сторонними сервисами

1.6.1 API погоды

Для получения погоды используется API сервиса «Яндекс Погода». Этот сервис был выбран из-за инновационной технологии Meteum, которая

использует машинное обучение, множество данных с разных метеостанций, а также местоположение пользователя, чтобы составить наиболее точный прогноз. Причиной выбора сервиса послужили также удобство использования и то, что изначально сервис проектировался для работы в России, как и данное приложение.

2 РАЗРАБОТКА СТРУКТУРЫ ПРИЛОЖЕНИЯ

2.1 Описание архитектуры программы

Приложение реализует архитектуру клиент-сервер. Код приложения разделен на три слоя: presentation, domain, data. Presentation отвечает за графический интерфейс, domain за представление данных, data – за их обработку и передачу. Экраны приложения разработаны по архитектуре MVVM (Model-View-ViewModel) с использованием классов ViewModel и LiveData из набора библиотек Android Architecture Components.

Паттерн MVVM (Model-View-ViewModel) был предложен компанией Microsoft в 2005 году Джоном Госсманом. В 2017 году компания Google предложила рекомендуемую архитектуру android приложений с использованием паттерна MVVM. Для этого был создан набор архитектурных компонентов (Android Acrhitecture Components – AAC), в который входят компоненты ViewModel и LiveData. Архитектура MVVM показана на рисунке 2.1.

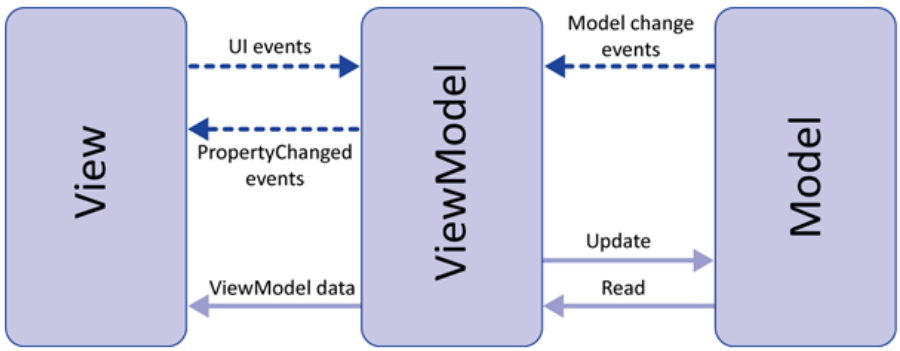

Рисунок 2.1 – Архитектура MVVM.

Шаблон MVVM делится на три части:

1. *Модель* (англ. *Model*) (так же, как в классической MVC) представляет собой логику работы с данными и описание фундаментальных данных, необходимых для работы приложения.
2. *Представление* (англ. *View*) - графический интерфейс (окна, списки, кнопки и т. п.). Выступает подписчиком на событие изменения

значений свойств или команд, предоставляемых Моделью Представления. В случае, если в Модели Представления изменилось какое-либо свойство, то она оповещает всех подписчиков об этом, и Представление, в свою очередь, запрашивает обновлённое значение свойства из Модели Представления. В случае, если пользователь воздействует на какой-либо элемент интерфейса, Представление вызывает соответствующую команду, предоставленную Моделью Представления.

3. *Модель Представления* (англ. *ViewModel*) - с одной стороны, абстракция Представления, а с другой — обёртка данных из Модели, подлежащие связыванию. То есть, она содержит Модель, преобразованную к Представлению, а также команды, которыми может пользоваться Представление, чтобы влиять на Модель. Стоит заметить, что отличительной особенностью ViewModel является использование паттерна Наблюдатель для уведомления View об изменении данных.

Такая структура экрана позволяет отделить логику обработки модели от логики её отображения. При этом каждый компонент слабо связан друг с другом, что позволяет легко вносить изменения и тестировать каждую часть по отдельности.

На официальном сайте для Android разработчиков демонстрируется типичная архитектура приложения с использованием MVVM. Диаграмма этой архитектуры представлена на рисунке 2.2.

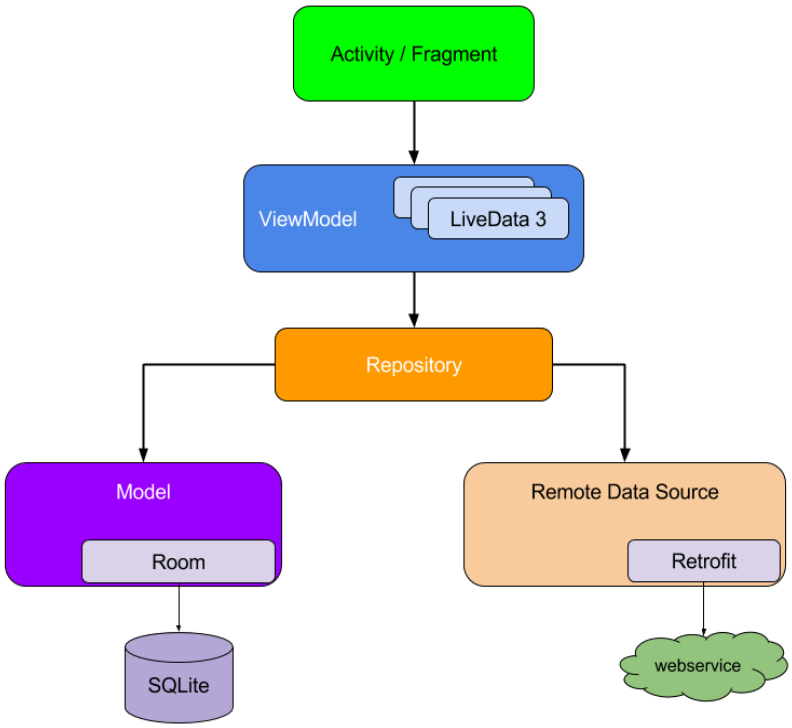

Рисунок 2.2 – Архитектура приложения с MVVM.

В качестве View выступает Activity или Fragment. Данные поступают во View из ViewModel с помощью компонентов LiveData, которые реализуют паттерн Наблюдатель, а также привязаны к жизненному циклу View, что избавляет программиста от необходимости вручную отписываться от наблюдаемого объекта (LiveData) при, например, скрытии экрана. В роли модели выступает Repository, который является абстракцией, инкапсулирующей источники данных. То есть репозиторий определяет, из какого места загрузить актуальную модель (база данных, кэш в ОЗУ или сервер).

Все данные в модели оборачиваются в т.н. реактивные объекты (из библиотеки RxJava), на которые подписывается ViewModel, используя функцию обратного вызова (англ. Callback). Затем полученные данные в callback'е передаются в соответствующий объект LiveData.

Приложение построено на базе парадигмы реактивного программирования. Реактивное программирование – это комбинация лучших идей из паттернов Наблюдатель (англ. Observer), Итератор (англ. Iterable) и функционального программирования. Суть этого подхода заключается в

представлении модели как потока данных, над которым можно производить различные преобразования в виде цепочек операций (применяемых функций). Библиотека RxJava предоставляет несколько реактивных оберток для данных (Observable, Single, Maybe, Completable…) и удобную абстракцию для выполнения фоновых операций в отдельных нитях, то есть использование паттерна Наблюдатель и управление стратегиями создания нитей. Таким образом, RxJava позволяет организовать асинхронную цепочку обработки данных перед отображением во View.

В приложении взаимодействие с сервером происходит в отдельном потоке и показ данных в специальном UI потоке. Это показано на примере экрана гардероба на рисунке 2.3.

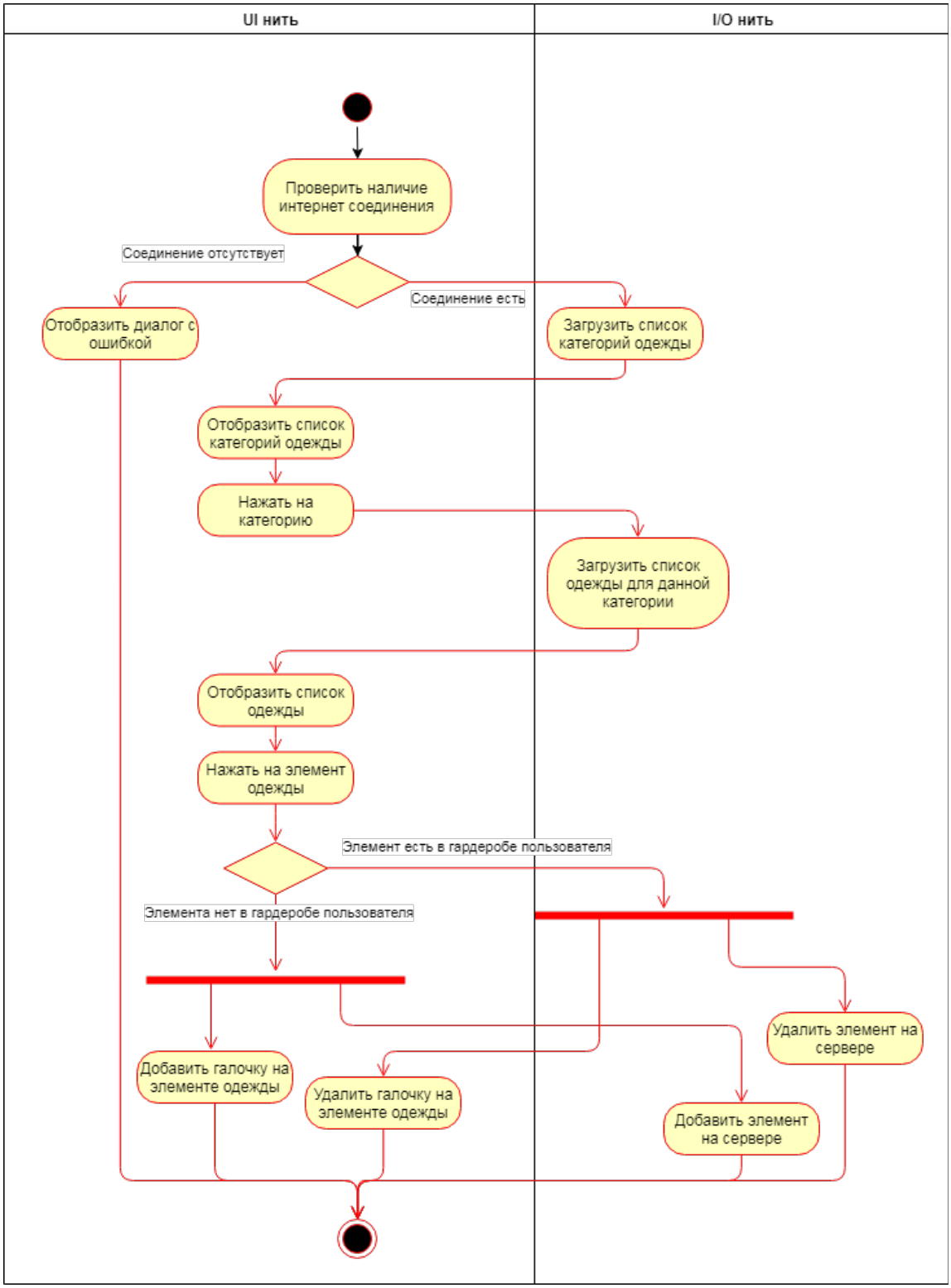

Рисунок 2.3 – Диаграмма деятельности экрана гардероба

Сервер написан на базе микросервисной архитектуры. Большинство программ пишутся в виде т.н. «монолита», то есть весь программный код находится в рамках одного проекта и запускается в одном процессе. Суть

микросервисной архитектуры в разделении монолита на отдельные сервисы по функциональному признаку. Каждый сервис запускается и работает независимо (в идеальном случае) и коммуницирует с другими сервисами посредством некоего протокола. Чаще в качестве протокола выступает http из-за большой распространенности и простоты использования. Преимуществом данного способа разработки заключается в возможности оптимизации нагрузки. Для этого производится т.н. оркестрация, то есть имеется некий прокси (написанный вручную или готовая утилита), который отслеживает количество запросов к сервисам и если какой-то сервис перегружен и не успевает обработать все запросы, то прокси запускает второй (третий, четвертый…) экземпляр этого же сервиса для распределения нагрузки. Далее этот прокси делит «поток» запросов между двумя сервисами поровну. Вторым достоинством микросервисов является возможность распределения штата разработки на несколько маленьких независимых команд, что позволяет уменьшить сложность в управлении процессом разработки и даже выбирать разные языки программирования для каждого сервиса (монолит тоже это позволяет, но это не так просто сделать). На рисунке 2.4 представлена структура взаимосвязей микросервисов, где направление стрелок указывает на то, куда отправляются http ответы. Следуя этому утверждению, можно понять, что MainService является связующим звеном между клиентом и всеми остальными сервисами.

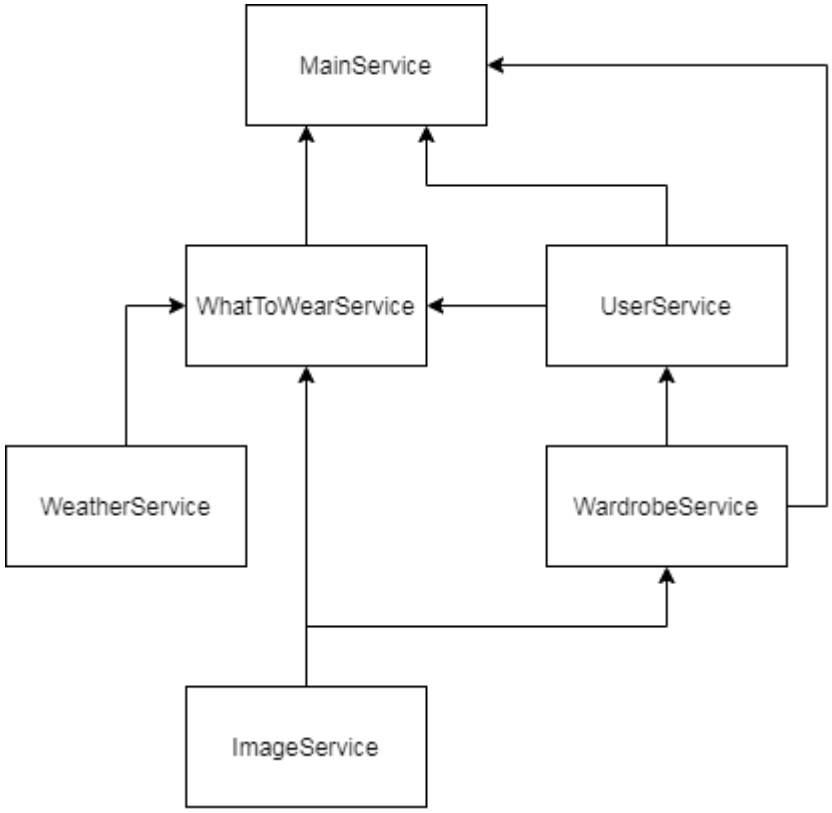

Рисунок 2.4 - Микросервисная архитектура сервера WannaDress

Структура сервисов следующая:

– **MainService -** Занимается перенаправлением всех http запросов на другие сервисы. Это дает единственную «точку входа» для клиентских приложений, а в перспективе можно добавить дополнительную обработку некоторых запросов, например, проверку авторизации;

– **UserService** – Выполняет CRUD (Create, Read, Update, Delete) операции с аккаунтом пользователя и выдает JWT (JSON Web Token) для авторизации;

– **WhatToWearService** – Самый важный компонент, который занимается подбором одежды и сбором данных для улучшения алгоритма этого подбора;

– **WardrobeService** - Выдает список категорий одежды и самих вещей вместе со ссылками на их изображения;

– **WeatherService** - Предоставляет данные о погоде;

– **ImageService** - Хранит статические изображения и генерирует изображение для человечка, то есть накладывает несколько изображений одежды поверх «манекена».

Для каждого сервиса существует файл ApiView, в котором содержатся обработчики http запросов, проверяющие валидность входных параметров, вызывающие нужные функции бизнес-логики и отправляющие ответ с результатом выполнения запроса. Все запросы к API начинаются с префикса “/api” (http://mydomain/api/auth), что дает возможность использовать одно и тоже доменное имя и для веб-сайта, и для сервера мобильного приложения. Часть запросов, которая работает с пользовательскими данными, требует JWT токен авторизации. За счет выборочного требования авторизации со стороны клиента можно обеспечить возможность пользоваться демо-функционалом без предоставления своих личных данных.

Диаграмма развертывания приложения показана на рисунке 2.5. На диаграмме видно, что все микросервисы находятся на одном сервере, а базы данных развернуты на отдельных машинах. Это позволит снизить нагрузку на сервер. Есть возможность перенести часть микросервисов на другой сервер, прописав при этом альтернативные ip и порты в конфигурации MainService. В теории можно сделать несколько экземпляров одного и того же сервиса для распределения нагрузки, но подобная оркестрация на данный момент не реализована.

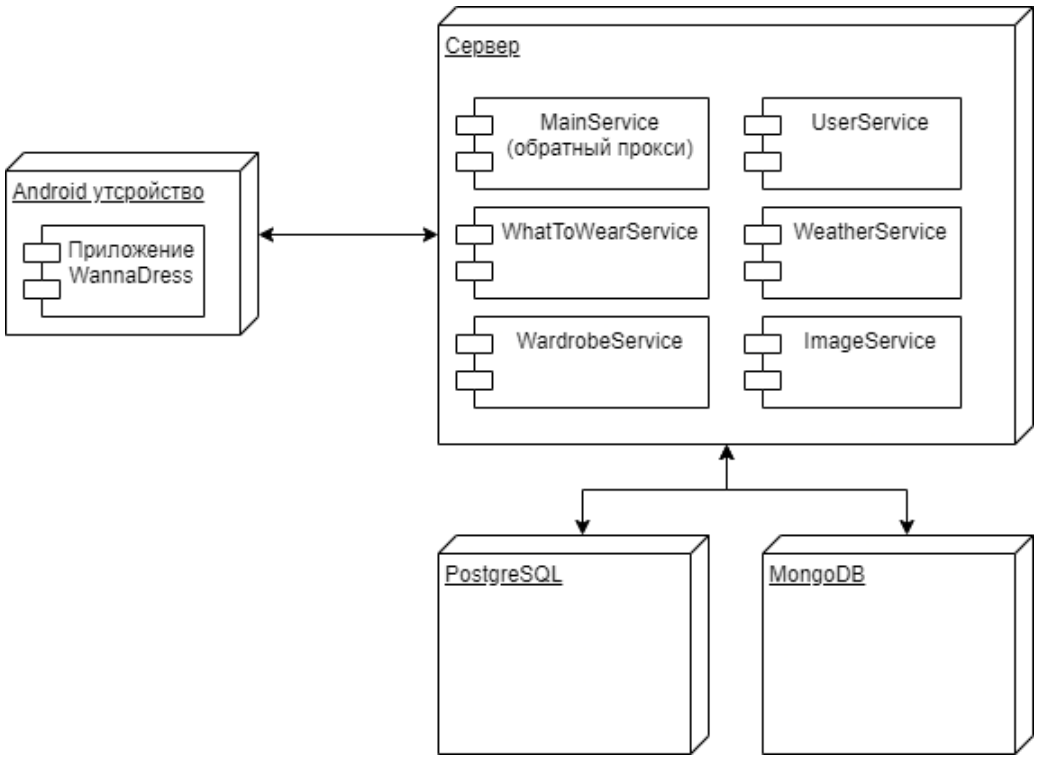

Рисунок 2.5 – Диаграмма развертывания

Взаимодействие между всеми компонентами системы показано на примере экрана гардероба на рисунке 2.6.

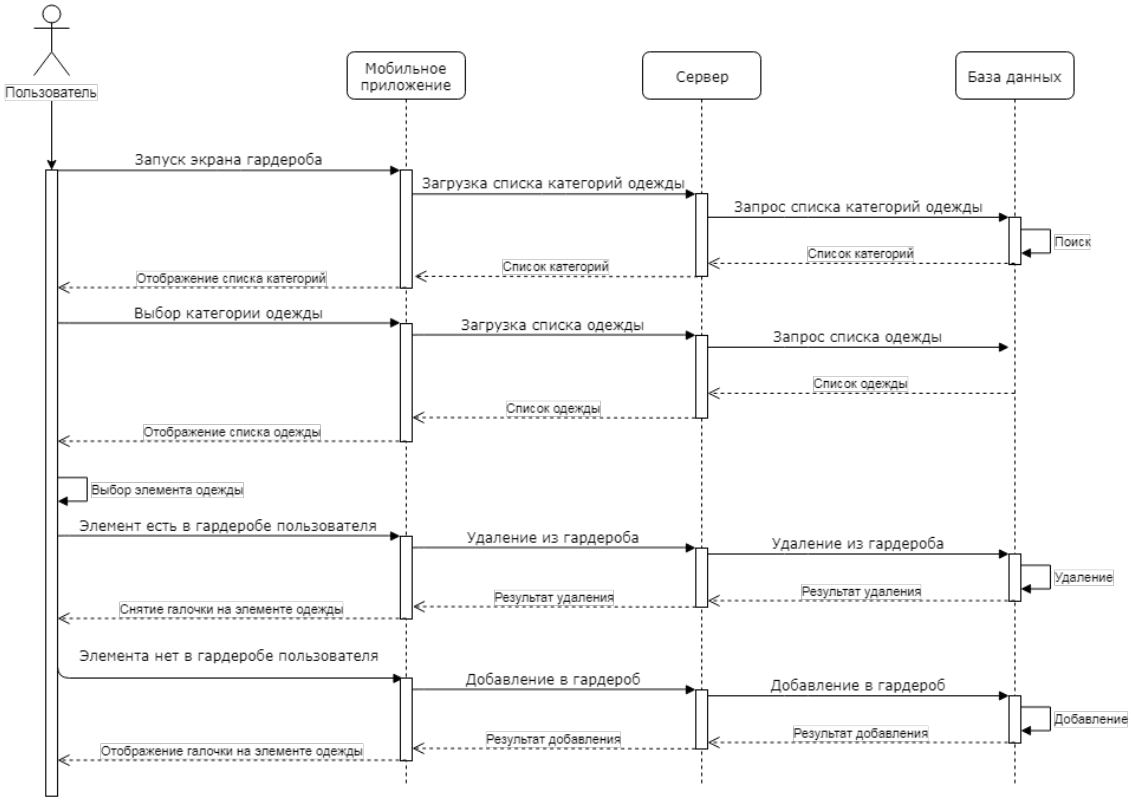

Рисунок 2.6 – Диаграмма последовательности экрана гардероба

Диаграмма вариантов использования мобильного приложения показана на рисунке 2.7.

Рисунок 2.7 – Диаграмма вариантов использования приложения

Варианты использования предполагают наличие одного актора – пользователя приложения.

2.2 Описание архитектуры базы данных

Для хранения данных, связанных с бизнес-логикой сервера, было выбрано использование реляционной базы данных. База содержит пять таблиц: clothes_categories, user, clothes, wardrobe, incompatible_categories. Структура каждой таблицы и связи между ними показаны на рисунке 2.8.

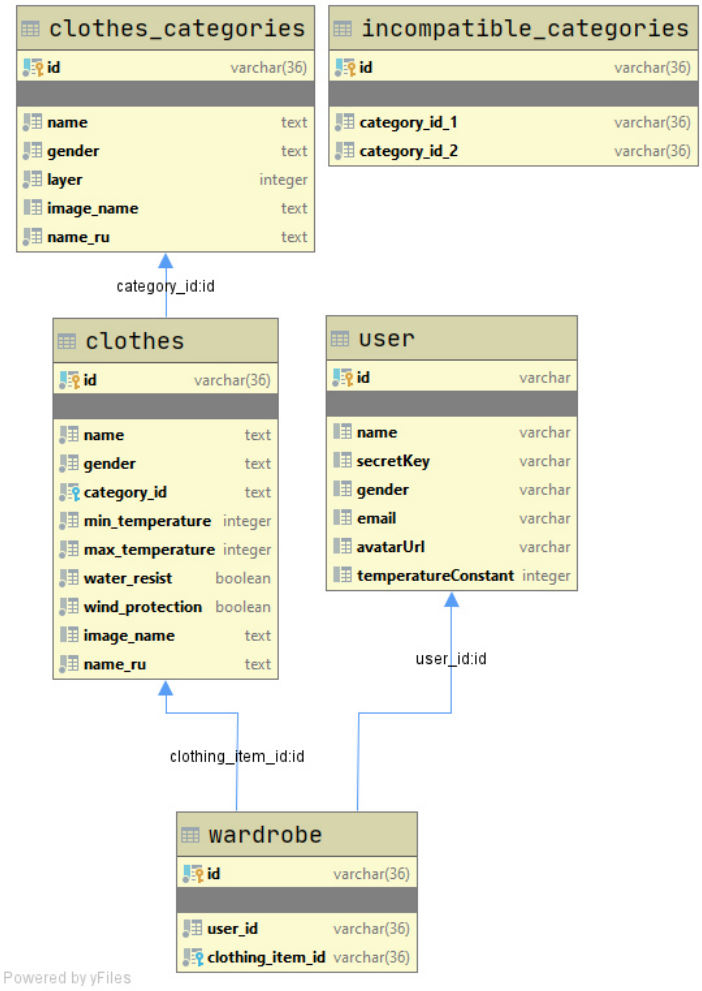

Рисунок 2.8 – ER диаграмма

С точки зрения идеи проекта самыми важными таблицами являются clothes_categories и clothes, которые, соответственно, хранят категории одежды и саму одежду. Группировка одежды по категориям позволяет упростить взаимодействие с массивом одежды с точки зрения клиента (Например, в мобильном приложении пользователь сначала видит список категорий, а по нажатию на категорию открывается список одежды для данной категории. Без классификации одежды клиент бы видел огромный список одежды, которым было бы сложнее пользоваться) и дает возможность накладывать бизнес-правила на подмножество вещей, как это делает поле layer, которое определяет порядок накладывания одежды друг на друга (майка должна быть под курткой и т.д.). Таблица с одеждой уже содержит данные, которые могут помочь при

выборе вещи для конкретных погодных условий, например, диапазон температуры, при которой комфортно носить эту одежду (min_temperature и max_temperature). Пример параметризованного SQL запроса одежды, который возвращает вещи для мужского пола:

*SELECT * FROM clothes*

WHERE imageName IS NOT NULL

AND gender = 'male' OR gender = 'all'

В таблице для пользователей помимо информации, идентифицирующей его, стоит отметить поле temperature_constant, которое влияет на подбор одежды для конкретного пользователя. По сути это число, которое сдвигает допустимый диапазон температуры для подбора одежды, ведь каждый человек воспринимает погоду по-своему. Эта константа вычисляется на основе опроса пользователей о степени комфортности предыдущего результата работы алгоритма подбора. Также результаты опроса сохраняются в отдельную СУБД для последующего обучения нейронной сети.

Таблица гардероба не представляет собой ничего необычного и просто содержит отношение между пользователем и одеждой, которая добавлена в его гардероб. По умолчанию гардероб пользователя пуст и алгоритм игнорирует предпочтения пользователя, основываясь только на характеристиках одежды и текущих погодных условиях. Если же в гардероб добавлена подходящая вещь для данной погоды, то алгоритм отдает предпочтение этому элементу одежды при формировании конечного набора вещей. Пример SQL запроса на получение гардероба пользователя:

*SELECT * FROM wardrobe WHERE user_id = 'some id'*

Таблица incompatible_categories содержит пары несовместимых категорий одежды. Например, не получится носить платье и юбку одновременно (по крайней мере так не принято).

Сбор массива информации для машинного обучения происходит в NoSQL базу данных, то есть у входных данных нет четко определенной

структуры, но на данный момент схема выглядит примерно, как следующий JSON документ:

{

“user_opinion”: “hot”,

"clothes": […],

"weather": {…}

}

User_opinion означает мнение конкретного пользователя о наборе одежды, в данном случае ему было жарко (“hot”). Clothes означает, собственно, набор одежды. Weather содержит погоду, для которой была подобрана одежда.

На рисунке 2.9 показаны классы, которые представляют информацию из таблиц базы данных.

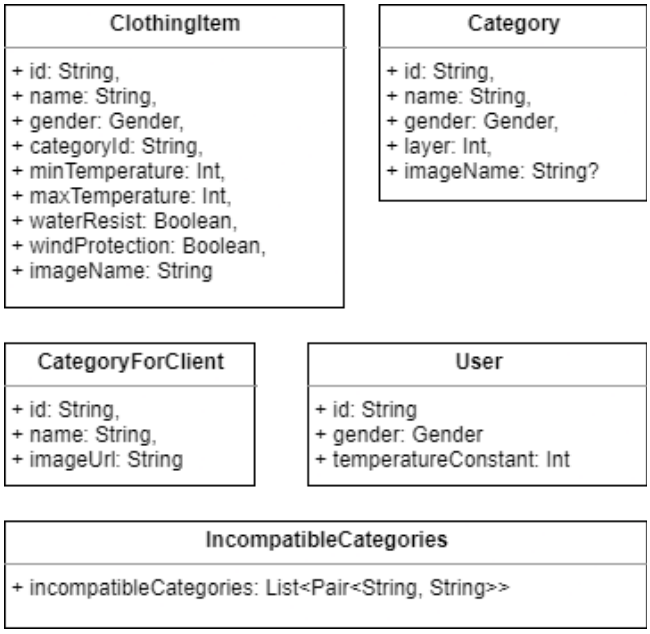

Рисунок 2.9 – Диаграмма классов

Некоторые модели содержат меньше полей, если они не требуются для использования в программе напрямую, а, например, задействованы только в SQL запросах. Или есть урезанные модели с меньшим количеством полей, которые впоследствии будут отправлена клиенту.

3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

3.1 Реализация клиента

3.1.1 Описание работы экрана AuthActivity

Экран на данный момент нужен для получения токена авторизации Google, чтобы отправить его на сервер, получить данные пользователя и авторизоваться в приложении. После авторизации сервер присылает JSON с данными пользователя, которые сохраняются локально в shared preferences (хранилище типа ключ-значение). Код авторизации:

```
val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken(googleAuthClientId)
        .requestEmail()
        .build()
val signInClient = GoogleSignIn.getClient(this, gso)
startActivityForResult(signInClient.signInIntent, RC_SIGN_IN)
```

3.1.2 Описание работы экрана MainActivity

Это самый главный экран, который всегда запускается при старте приложения. Вначале проверяется, авторизован ли пользователь, если нет, то открывается экран AuthActivity. Иначе начинается инициализация трех фрагментов («подэкранов»): WeatherFragment, CategoriesFragment, SettingsFragment. Переключение между фрагментами происходит с помощью элемента графического интерфейса BottomNavigationView.

WeatherFragment содержит информацию о погоде, прогнозе и наборе одежды. Представляет собой список (RecyclerView) с двумя типами элементов: элемент, содержащий текущую погоду и человечка в одежде и элемент прогноза погоды. На данный момент под текущей погодой показывается восемь элементов прогноза погоды. Но если нужно будет изменить количество элементов прогноза, то достаточно поменять ответ сервера, и клиент

подстроится под новое количество без каких-либо изменений в коде работы списка.

На WeatherFragment можно посмотреть подробный список одежды для текущей погоды, если нажать на человечка. Тогда покажется BottomSheetDialogFragment, который представляет собой диалоговое окно, но которое не появляется поверх экрана, а «выезжает» снизу и закрывает примерно половину экрана. Затем это окно можно дотянуть до верха экрана, чтобы диалог полностью закрыл весь экран, либо потянуть вниз, чтобы закрыть.

CategoriesFragment отображает список категорий одежды и в конце списка находится элемент, по нажатию на который открывается диалог обратной связи. Если нажать на категорию одежды, то поверх данного фрагмента откроется CategoryFragment, который уже будет показывать список одежды для данной категории и снова кнопку для обратной связи. На CategoryFragment можно добавлять или удалять вещи в гардеробе, потом эта информация отправляется на сервер. Диаграмма классов, отображающая сущности для работы списка категорий одежды, представлена на рисунке 3.1. На ней видно, что список имеет три вида элементов: категория, кнопка обратной связи и реклама. Рекламный элемент на данный момент не используется.

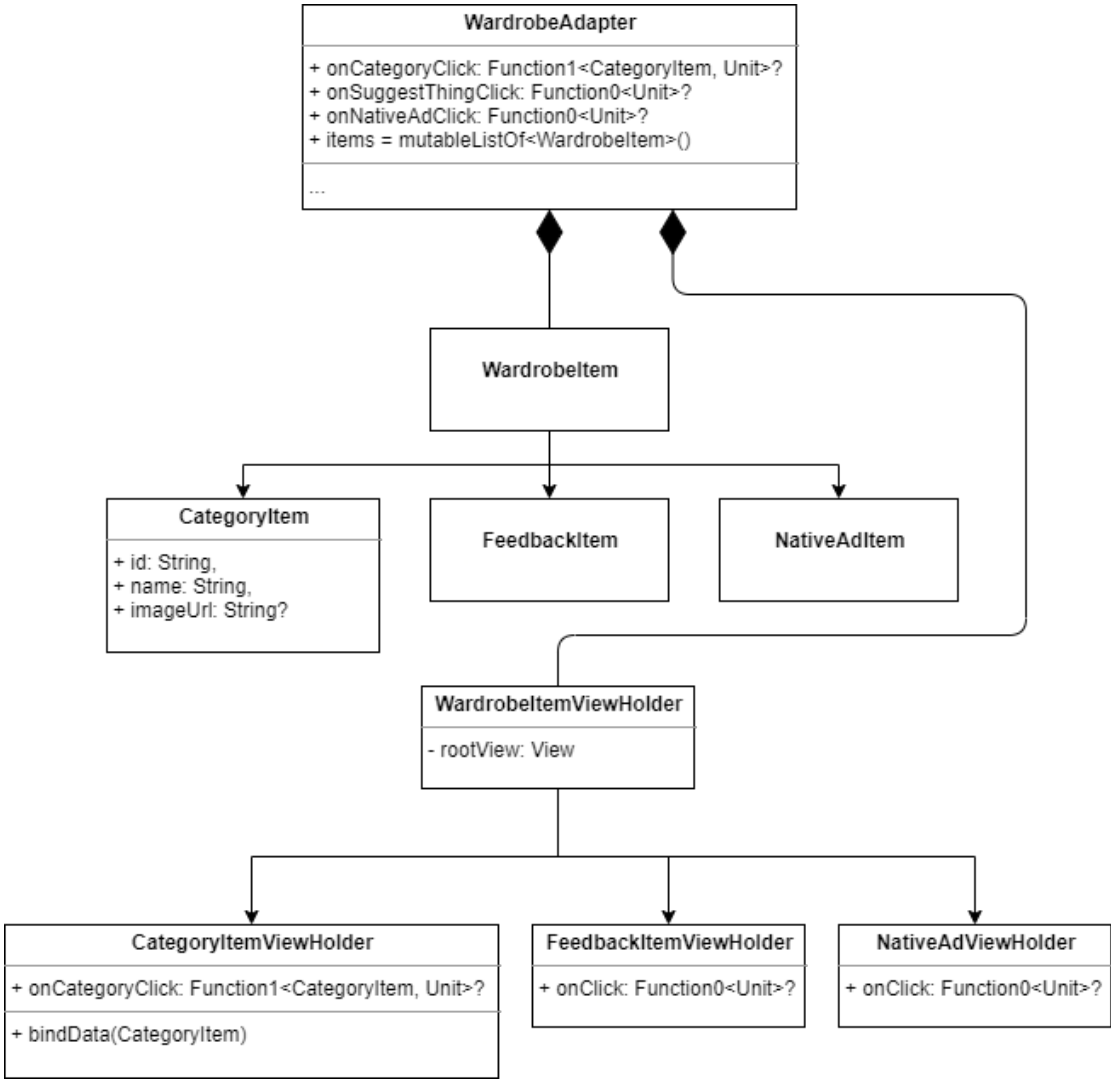

Рисунок 3.1 – Диаграмма классов списка CategoriesFragment

SettingsFragment содержит информацию о пользователе и настройки. Можно изменить пол пользователя по нажатию на «радио-кнопку» или включить/отключить уведомления. Напоминание о том, чтобы пользователь зашел в приложение выставляется с помощью AlarmManager в восемь часов утра. Код установки повторяющегося уведомления:

```
val alarmManager = context.getSystemService(Context.ALARM_SERVICE) as AlarmManager
    alarmManager.setRepeating(
        AlarmManager.RTC,
        calendar.timeInMillis,
```

```
            TimeUnit.DAYS.toMillis(1),
            pendingIntent
        )
```

3.2 Реализация сервера

3.2.1 Описание работы MainService

Сервис выполняет обратное проксирование всех http запросов, которые на него поступают. В зависимости от пути запроса сервис решает, куда перенаправить текущий запрос. Прокси-сервер – это сервер, который перенаправляет все поступающие на него запросы на сторонние сервера и создает видимость для клиента, что ответы формируются непосредственно на прокси-сервере. Обратный прокси-сервер работает также, только ретранслирует запросы на ограниченный набор дочерних серверов. Функция проксирования получает все входные параметры (тело, заголовки и т.д.) и просто пересылает их на нужный адрес. Вот как выглядит эта функция:

```
fun Route.reverseProxy(baseUrl: String) {
    handle {
        val response = httpClient.request<HttpResponse> {
            method = call.request.httpMethod
            url(baseUrl + call.request.uri)
             headers.appendAll(call.request.headers.filter { header, _ -> header != "Content-Length" })
             call.receiveOrNull<ByteArray>()?.let { receivedBody -> this.body = receivedBody }
        }
        call.respond(response.status, response.readBytes())
    }
}
```

Некоторые точки предварительно проверяются на наличие заголовка Authorization с JWT токеном авторизации. JSON Web Token (JWT) – это

стандарт для создания токенов доступа. Токен представляет собой некую закодированную последовательность символов, содержащую полезную нагрузку – служебную информацию.

3.2.2 Описание работы WeatherService

Данный микросервис обрабатывает только один запрос на погоду, который возвращает текущую погоду и почасовой прогноз на 24 часа. Сервис обращается с API Яндекс.Погоды, получает ответ, преобразует полученные данные к собственному формату и отсылает ответ клиенту. Код выглядит так:

```
suspend fun weatherByCriteria(latitude: Double, longitude: Double, language: Language): Weather {
    return httpClient.get<YandexWeatherResponse> {
        url("https://api.weather.yandex.ru/v1/forecast")
        header("X-Yandex-API-Key", yandexWeatherApiKey)
        url {
            parameter("lat", latitude)
            parameter("lon", longitude)
            parameter("lang", language.yandexWeatherValue)
            parameter("limit", 1) // this maybe doesn't work now
            parameter("hours", true)
            parameter("extra", true) // for extra info about precipitation
        }
    }.toWeather(language)
}
```

Наличие собственного формата данных позволяет без лишней траты времени либо заменить API для получения погоды, либо использовать несколько сторонних API и как-то усреднять полученные данные о погоде.

3.2.3 Описание работы WardrobeService

Этот сервис отвечает за выдачу категорий одежды, выдачу одежды, выдачу списка несовместимых категорий, добавление вещей в гардероб пользователя и удаление вещей из гардероба. Все точки, которые связаны с получением данных, используются сервисом WhatToWearService во время подбора одежды, либо клиентом при просмотре доступных вещей. Добавление и удаление вещей происходит только со стороны клиента. В качестве примера далее приведена функция удаления вещи из гардероба пользователя:

```
fun deleteClothingItemFromWardrobe(userId: String, itemId: String) {
    transaction {
            UserDAO.findById(userId) ?: throw NotFoundException("uid = $userId")
        ClothesDAO.findById(itemId) ?: throw NotFoundException("itemId = $itemId")
            WardrobeDAO.find { (WardrobeTable.userId eq userId) and (WardrobeTable.clothingItemId eq itemId) }
            .forEach { it.delete() }
    }
}
```

3.2.4 Описание работы UserService

Пользовательский микросервис отвечает за авторизацию и CRUD (Create Read Update Delete) операции с пользователем. Авторизация происходит следующим образом:

- Клиент производит авторизацию через аккаунт Google на мобильном устройстве
- Мобильное приложение посылает полученный google-токен авторизации на сервер

– Сервер отсылает этот токен на API Google и получает личные данные пользователя (такие, как имя, почта и т.д.)
– Сервер генерирует секретный ключ, который используется при генерации JWT токена
– Генерируется JWT токен и отправляется клиенту для дальнейшего использования

Чтобы проверить токен авторизации, сервис MainService посылает сервису UserService специальный запрос на валидацию токена при каждой попытке сделать запрос, требующий наличия авторизации.

3.2.5 Описание работы WhatToWearService

Сервис выполняет функцию подбора одежды и сбора данных для дальнейшего совершенствования алгоритма подбора с помощью машинного обучения. Обработчик запроса на сбор данных выполняет проверку входного JSON документа на соответствие формату и в случае успеха записывает полученную информацию в базу данных MongoDB.

Алгоритм подбора одежды с учетом погодных условий работает примерно следующим образом:

– обработчик http запроса извлекает входные данные, которые содержат информацию о погоде и пользователе;

– происходит загрузка гардероба пользователя, всей доступной одежды, всех доступных категорий и набора несовместимых категорий одежды;

– вычисляется средняя температура среди текущей погоды и всех прогнозов;

– происходит фильтрация вещей пользователя по признаку соответствия средней температуре, полученной на предыдущем шаге;

– случайным образом выбирается одна вещь из каждой категории (кроме аксессуаров, они отбираются отдельно);

– если есть вещи, категории которых несовместимы, то случайным образом выбирается одна из конфликтующих вещей;

– производится отбор аксессуаров по погодным условиям;

– формируется общий набор вещей и аксессуаров, которые подходят для данных погодных условий.

Так выглядит функция для вычисления средней температуры:

```
private fun calculateAverageTemperature(weather: Weather): Int {
    val forecastTemperatures = weather.forecasts.map { forecast ->
                    forecast.forecastsForHour.map { forecastForHour ->
forecastForHour.feelsLikeTemperature }
    }.flatten().take(HOURS_IN_A_HALF_OF_A_DAY)
                                                                return
forecastTemperatures.plusElement(weather.currentWeather.feelsLikeTemper
ature).average().roundToInt()
}
```

3.3 Руководство пользователя

Приложение состоит из нескольких экранов, использование которых рассмотрено далее.

1. **Экран авторизации.** Для того, чтобы начать использование приложения, нужно авторизоваться с помощью Google аккаунта. Для этого нужно нажать единственную на экране кнопку и в появившемся диалоге выбрать подходящий аккаунт. Если у вас только один аккаунт на устройстве, то диалог выбора может и не появиться (зависит от особенностей ОС). Экран показан на рисунке 3.2.

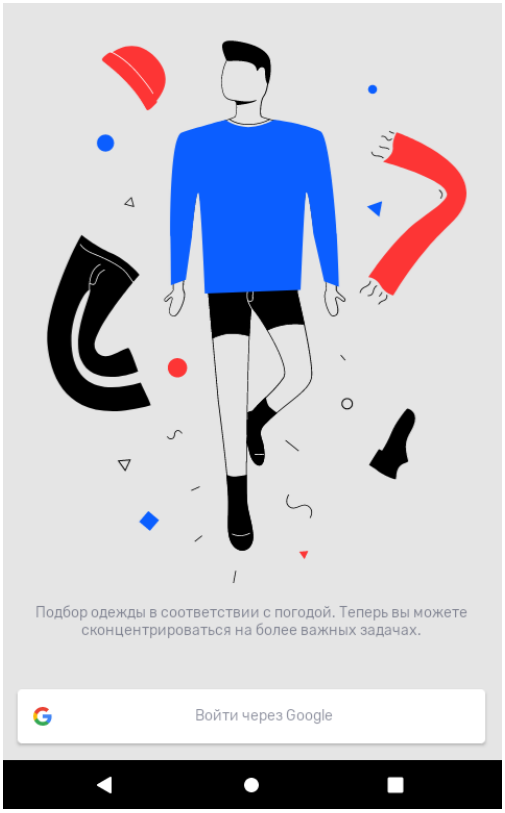

Рисунок 3.2 - Экран авторизации

2. **Главный экран.** Изображен на рисунке 3.3. На этом экране отображается основная информация:

1) текущая температура;
2) текущая погода в форме иконки;
3) описание текущей погоды;
4) город, в котором находится пользователь;
5) небольшая таблица с прогнозом погоды через каждые 2 часа;
6) человечек в одежде, предложенной алгоритмом на основе погодных условий.

Чтобы обновить информацию на главном экране, нужно потянуть сверху вниз. Фоны на главном экране меняются в зависимости от текущей погоды. По нажатию на человечка можно открыть подробный список предложенной одежды, который появляется поверх всего экрана.

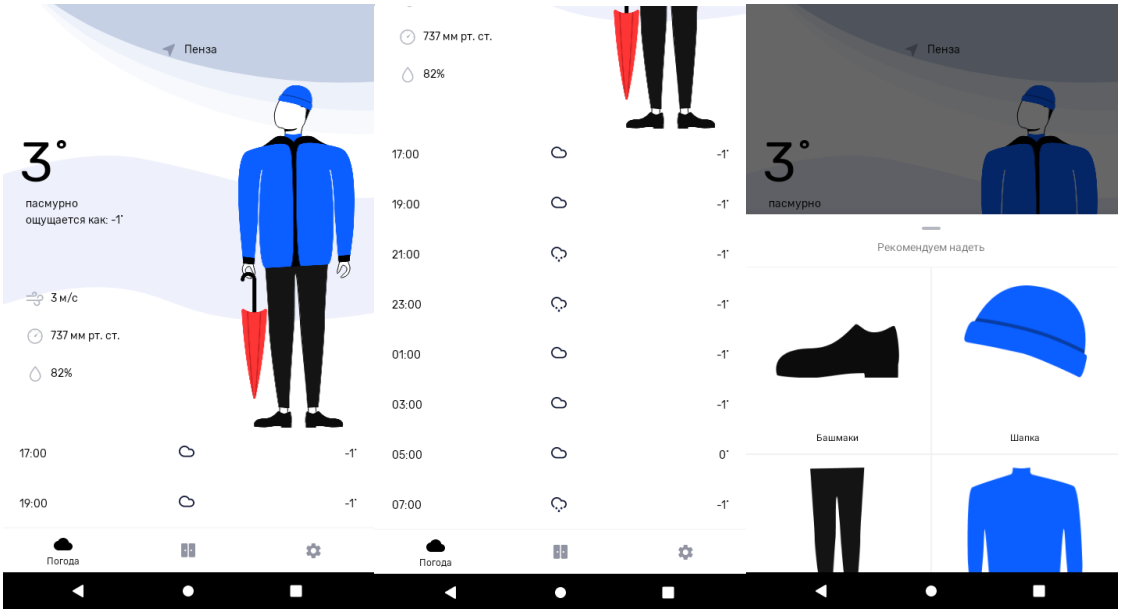

Рисунок 3.3 - Главный экран

3. **Экран настроек.** Здесь можно изменить пол пользователя, выйти из аккаунта и настроить уведомление, в котором будет содержаться важный текст, например, «Не забудь взять зонт, сегодня ожидается ливень». Для выполнения этих функций на экране присутствуют соответствующие кнопки и переключатели, показанные на рисунке 3.4, использование которых интуитивно понятно.

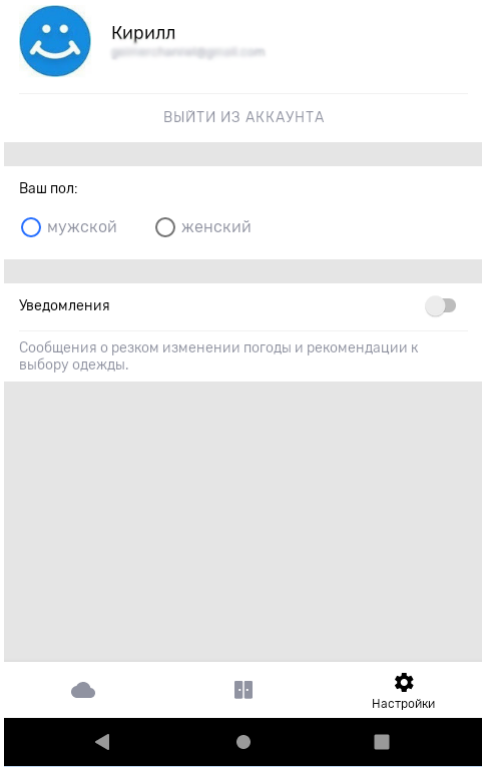

Рисунок 3.4 - Экран настроек

4. **Экран гардероба.** Это список категорий одежды, в каждой категории содержится список типов одежды для данной категории. Все состояния экрана показаны на рисунке 3.5.

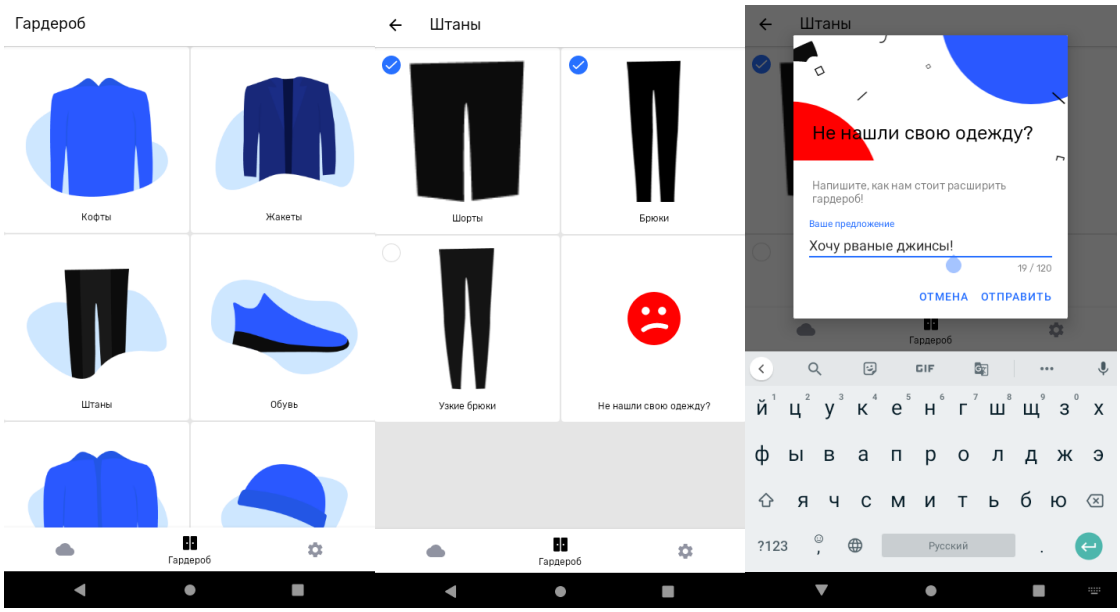

Рисунок 3.5 - Экран гардероба

На каждом типе одежды можно поставить галочку. Эта информация отправляется на сервер и начинает учитываться при подборе вещей. Так же есть кнопка с «грустным лицом», по нажатию на которую открывается диалог обратной связи, где можно указать, какую одежду стоит добавить в приложение.

4 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. Чаще всего тесты разделяют на 3 вида:

– модульное тестирование – тестирование отдельных функций на программном уровне;

– интеграционное тестирование – тестирование корректности работы отдельных компонентов системы вместе;

– системное тестирование - тестирование на уровне пользовательского интерфейса.

4.1 Тест-кейсы

Тест-кейсы представлены в таблице 1.

Таблица 1 – Тест-кейсы

№ теста	Описание теста	Результат тестирования
1	Изменение пола пользователя в настройках	Экран погоды должен перезагрузить информацию с другим изображением человечка. В базе данных должно обновиться поле gender
2	Добавление вещи в гардероб пользователя	В таблице базы данных wardrobe появляется соответствующая запись
3	Попытка обновить данные на экране с погодой (WeatherFragment) при отсутствии интернет подключения	Показывается диалоговое окно с сообщением об ошибке
4	Включение уведомления на экране настроек	Уведомление с напоминанием зайти в приложение показывается в восемь утра

4.2 Результаты тестирования

На рисунке 4.1 показан экран настроек, на котором выбран мужской пол пользователя, и главный экран, на котором отображен человечек мужского пола. На рисунке 4.2 видно, что в базе данных поле gender равно male, то есть данные в базе совпадают с данными на экране настроек.

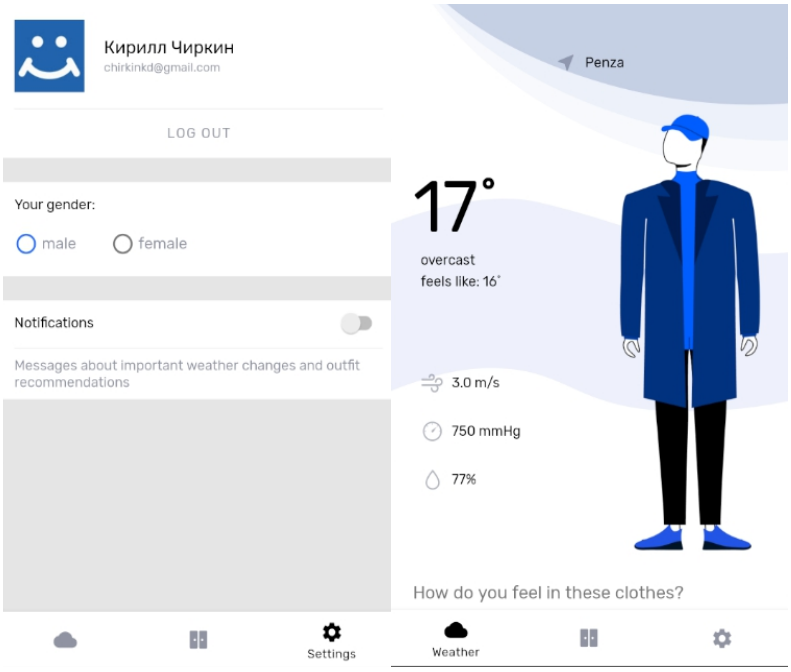

Рисунок 4.1 – Экран настроек и экран гардероба (мужской пол)

id	190a3df9-81fb-40cd-afee-446f85e74bfe
name	Кирилл Чиркин
secretKey	eed0bed0bbdbbaae0dde000abea0edadbbbb
gender	male
email	chirkinkd@gmail.com
avatarUrl	https://lh3.googleusercontent.com/a-/AOh1…
temperatureConstant	0

Рисунок 4.2 – Запись пользователя в базе данных (мужской пол)

Рисунок 4.3 демонстрирует изменение настроек пола пользователя на женский и соответствующее изменение человечка на главном экране. В базе данных поле gender изменилось на female, что показано на рисунке 4.4.

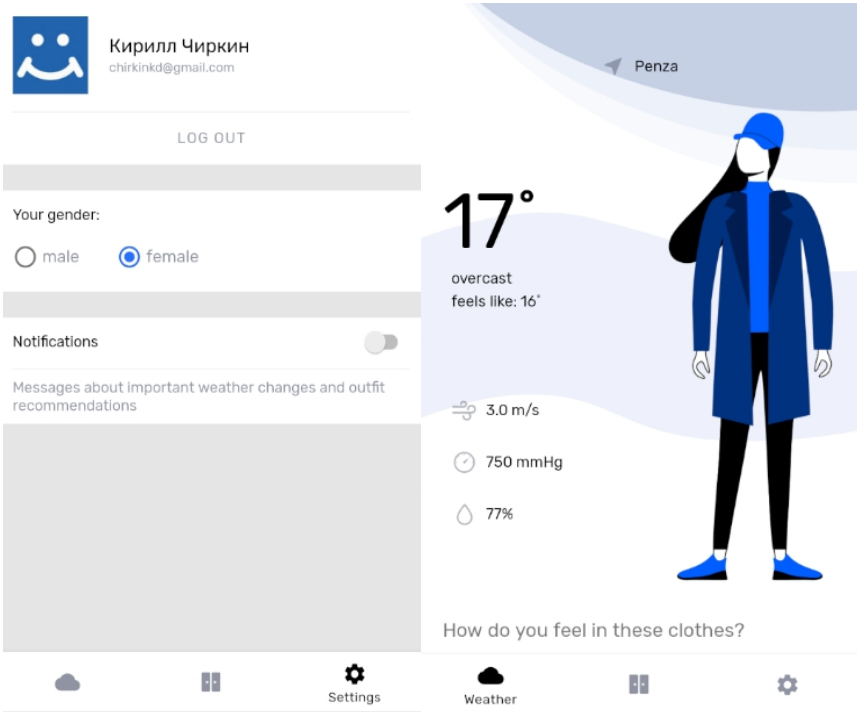

Рисунок 4.3 – Экран настроек и экран гардероба (женский пол)

id	190a3df9-81fb-40cd-afee-446f85e74bfe
name	Кирилл Чиркин
secretKey	eed0bed0bbdbbaae0dde000abea0edadbbbb
gender	female
email	chirkinkd@gmail.com
avatarUrl	https://lh3.googleusercontent.com/a-/AOh1…
temperatureConstant	0

Рисунок 4.4 – Запись пользователя в базе данных (женский пол)

На рисунке 4.5 на экране гардероба открыт раздел толстовок и показан предлагаемый набор одежды. Ни один элемент из категории толстовок не выбран, поэтому алгоритм берет любой элемент одежды, который подходит по погоде, в данном случае это водолазка (turtleneck). В таблице гардероба нет записей с идентификатором рассматриваемого пользователя, что и показано на рисунке 4.6.

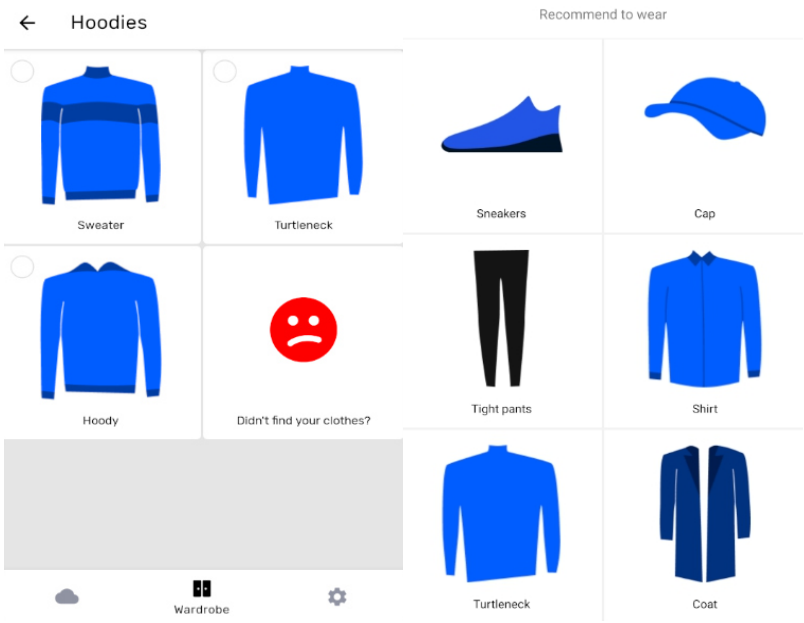

Рисунок 4.5 – Гардероб пользователя и предложенная одежда

	id	user_id	clothing_item_id
1	f08a4df8-4795-42c…	2336fe75-f42b-4fe1…	1d496aa6-13a1-43dd-a7b4…
2	45e99cb7-87cb-4d4…	2336fe75-f42b-4fe1…	e5d12bbe-6681-484a-aed9…
3	fe681b07-30ee-40f…	2336fe75-f42b-4fe1…	d7e8909a-e66a-429e-b452…
4	47c1fd4c-21bf-490…	c6a914e1-1d8b-42ef…	3949577f-f350-40c6-9e07…

Рисунок 4.6 – Таблица гардероба в базе данных

После добавления толстовки в гардероб пользователя алгоритм начинает отдавать предпочтение этому элементу одежды, поэтому в предлагаемом наборе появляется толстовка, что видно на рисунке 4.7. Рисунок 4.8 показывает, что появилась новая запись в таблице гардероба.

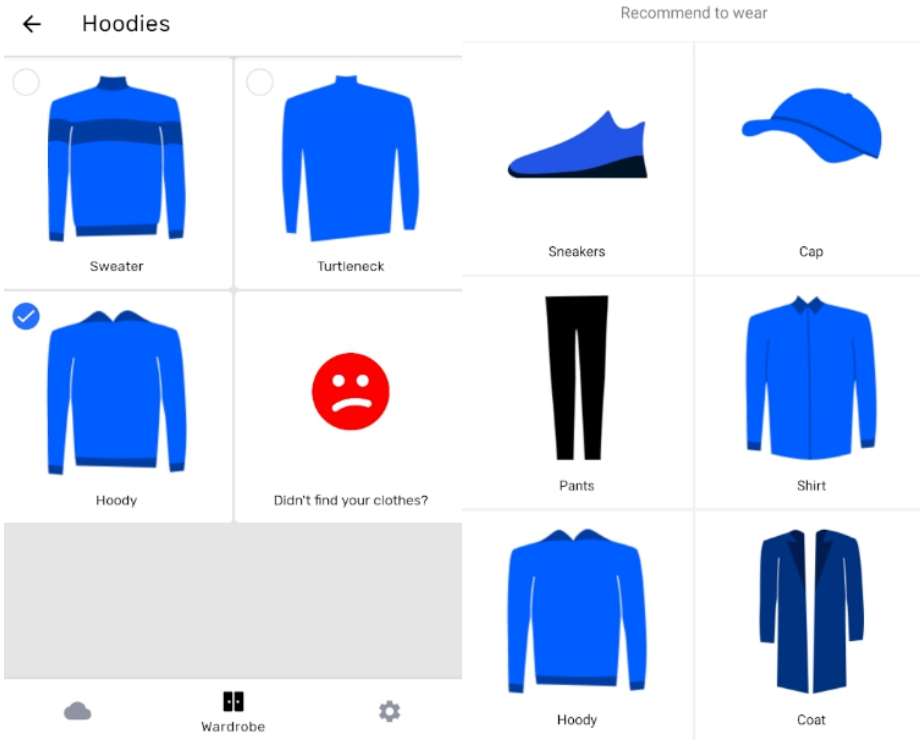

Рисунок 4.7 – Добавление толстовки и предложенная одежда

	id	user_id	clothing_item_id
1	f08a4df8-4795-42c…	2336fe75-f42b-4fe1…	1d496aa6-13a1-43dd-a7b4…
2	45e99cb7-87cb-4d4…	2336fe75-f42b-4fe1…	e5d12bbe-6681-484a-aed9…
3	fe681b07-30ee-40f…	2336fe75-f42b-4fe1…	d7e8909a-e66a-429e-b452…
4	47c1fd4c-21bf-490…	c6a914e1-1d8b-42ef…	3949577f-f350-40c6-9e07…
5	37fe2552-2d21-4a9…	190a3df9-81fb-40cd…	07c36a19-8da1-4fb3-b56c…

Рисунок 4.8 – Таблица гардероба после добавления толстовки

На рисунке 4.9 демонстрируется обработка типичных ошибок в приложении. В данном случае показана попытка обновить данные на главном экране при отсутствии интернет соединения (включен «режим полета»). В ответ на это показывается диалоговое окно с сообщением об ошибке и кнопками «Повторить» и «Ок».

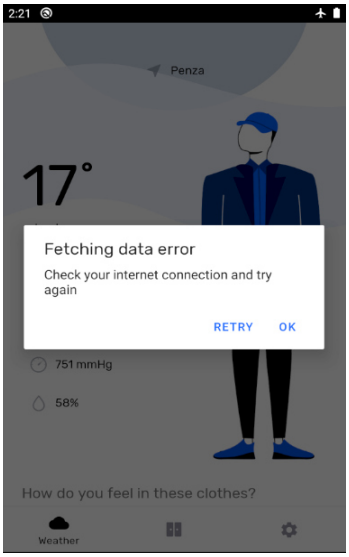

Рисунок 4.9 – Диалог с ошибкой загрузки данных

Рисунок 4.10 показывает экран настроек, на котором были включены уведомления, и шторку уведомлений с появившимся напоминанием об утреннем посещении приложения.

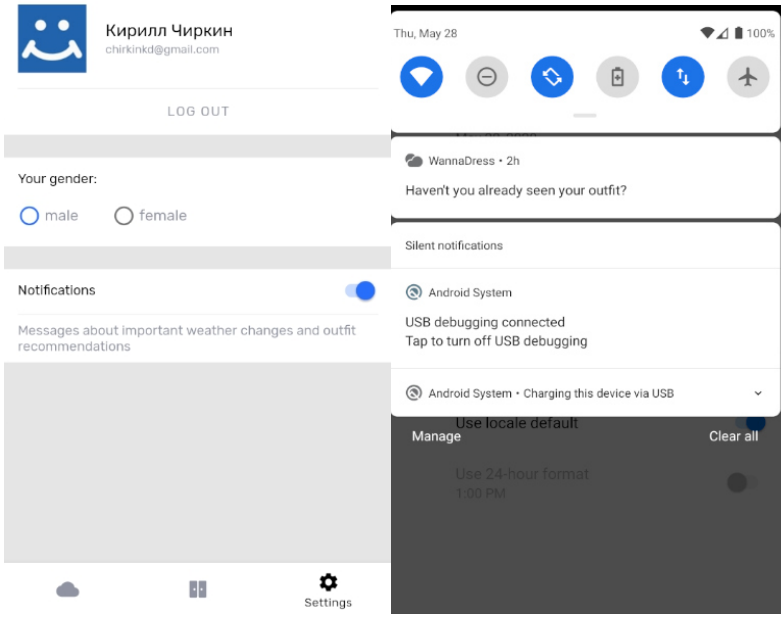

Рисунок 4.10 – Настройка уведомления и появившееся уведомление

Выводы на основе результатов тестирования:

– приложение корректно выполняет разработанный функционал,

- в ходе тестирования не удалось добиться выхода приложения из строя.

5 БЕЗОПАСНЫЕ УСЛОВИЯ ТРУДА ОПЕРАТОРА ЭВМ

5.1 Аспекты производственной безопасности при работе на ПК

ПК постоянно оказывает на человека вредное воздействие следующими поражающими факторами:

- психофизиологические факторы (утомление, монотонность работы, статические перегрузки и др.);
- электромагнитные излучения, исходящие от техники;
- шум издаваемый техникой и внешней средой;
- опасность поражения электричеством.

5.1.1 Психофизиологические факторы

Психофизиологические факторы определятся характером труда. Их делят на физические и нервно-психические перегрузки. Нервно-психические подразумевают под собой стресс, эмоциональную перегрузку, монотонность работы, умственное перенапряжение.

Для уменьшения влияния этих факторов необходимо применять следующие оптимальные режимы труда и отдыха в течение рабочего дня:

- контакт работника с монитором не должен превышать 50% всего рабочего времени;
- каждые 2 часа делать 15-минутные перерывы при обычной работе за ПК, при интенсивной работе через каждый час;
- желательно использовать жидкокристаллические дисплеи с высокой разрешающей способностью и комфортным размером экрана;
- при работе с ПК использовать экранные фильтры для дисплеев, это значительно снизит утомляемость глаз.

Программист или сотрудник работающий с ПК, основную часть рабочего времени проводит сидя за столом. Известно, что рабочая поза

значительно влияет на качество работы человека. Основные требования к рабочим местам при выполнении работы сидя приведены в ГОСТ 12.2.033-78 «ССБТ. Рабочее место при выполнении работ сидя. Общие эргономические требования» [14]. Необходимо чтобы рабочее место было оборудовано так, чтобы исключать неудобные позы и длительные статические состояния тела.

5.1.2 Защита от излучений

Основная причина развития охраны здоровья людей, работающих с ПК – это электромагнитные излучения. В ПК их источниками являются монитор (дисплей) и системный блок.

Исходя из данных об ЖК и ЭЛТ мониторах, можно сделать вывод, о том, что гораздо безопаснее использовать для работы ЖК мониторы. Однако надо не забывать, что в помещении скопление большого количества техники все равно может привести к неприятным последствиям. Поэтому размещение мониторов друг от друга желательно не менее расстояния вытянутой руки.

5.1.3 Оборудование рабочих мест с ПК

Требования к организации и оборудованию рабочего места оператора приведены в ГОСТ 12.2.032-78 [14]. Рабочий стул (кресло) должен быть подъемно-поворотным и регулируемым по высоте, углам наклона, а также расстоянию спинки до сидения. Рекомендованы следующие наклоны углов расположения монитора:

- для работы только стоя угол должен составлять 60 градусов;
- для работы как сидя, так и стоя – 45 градусов;
- для работы только сидя – 30 градусов.

5.1.4 Электробезопасность

Помещения для работы с компьютерной техникой и другими устройствами небезопасны, так как электрический ток представляет опасность для человека. В ГОСТ 12.2.007-75 устанавливаются требования к безопасности,

предотвращающие или уменьшающие до допустимого уровня воздействие на человека электрического тока.

Основные меры защиты от поражения электрическим током при работе с компьютером:

- защитные заземления;
- зануление;
- защитное отключение.

Для предотвращения выхода из строя компьютера, возникновения пожара или возможности поражения электрическим током необходимо соблюдать следующие меры предосторожности:

- запрещается во время работы компьютера отключать и подключать разъемы соединительных кабелей;
- запрещаются любые вмешательства в корпус системного блока до полного отключения от электропитания;
- запрещается разбирать монитор и пытаться самостоятельно устранять неисправности;
- запрещается использовать в качестве заземления водопроводные и газовые трубы, радиаторы и другие узлы парового отопления;
- Запрещается закрывать вентиляционные отверстия на корпусе системного блока и монитора во избежание перегрева элементов.

5.1.5 Освещение рабочего места

Для рабочего места программиста должно быть правильное освещение. Если работа подразумевает и работу с ПК и работу с бумагами, то на столе должна присутствовать дополнительная настольная лампа для подсветки документов.

Относительно яркого окна монитор следует располагать так, чтобы окно от него находилось сбоку. Так как если монитор расположить на фоне окна, то свет из него будет давать дополнительную нагрузку на глаза, а если окно будет

за спиной работающего, то изображение на мониторе будет очень тусклым, что тоже потребует дополнительного зрительного сосредоточения.

Обеспечение достаточно равномерного распределения яркости рабочей поверхности (светлый потолок, светлые стены и производственное оборудование) способствует созданию комфортного пространства в поле зрения работающих.

На рабочей поверхности должны отсутствовать резкие тени. Очень вредны подвижные тени.

В поле зрения работников должна отсутствовать повышенная яркость светящихся поверхностей. Она может вызвать частичное ухудшение видимости объектов.

Любые сбои, колебания или мерцания освещенности приводят к значительному утомлению глаз. Зачастую с пульсацией освещенности связана особенность газоразрядных ламп, поэтому для таких помещений следует выбрать дугой тип ламп.

5.2 Пожарная безопасность

Рабочее место программистов, где расположено электрооборудование, легковоспламеняющиеся вещества (мебель, бумага и др.) относится к категории В (пожароопасная) пожарной опасности помещений.

В связи с этим помещение должно соответствовать нормативам по огнестойкости строительных конструкций, планировке зданий, этажности, оснащенности устройствами противопожарной защиты, установленным для этой категории помещений.

В связи с предотвращением возгорания, в помещении запрещается:

- хранить и использовать горючие жидкости;
- использовать не предусмотренные рабочим местом электронагревательные приборы;
- эксплуатировать повреждённые проводки;

- использовать поврежденные розетки и другое неисправное электрооборудование;
- применять открытый огонь;
- курить в помещении;
- нарушать правила эксплуатации ПЭВМ и оргтехники [15].

По окончании работы необходимо обесточить все электроприборы.

ЗАКЛЮЧЕНИЕ

В ходе работы над выпускной квалификационной работой разработано мобильное приложение «WannaDress» для операционной системы Android на языке Kotlin, предназначенное для отображения набора одежды в зависимости от погодных условий. Для приложения разработан http сервер на языке Kotlin. Получен неоценимый опыт в android разработке и серверном программировании.

В дальнейшем можно доработать приложение в плане стабильности и удобства использования, добавить монетизацию и выложить в магазин приложений Google Play. Следует расширить список одежды, продолжать совершенствовать алгоритм подбора и добавить формирование набора вещей на несколько дней вперед для разного местоположения.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Android developer [Электронный ресурс] https://developer.android.com
2. Чиркин К.Д. Современная архитектура монолитного android-приложения // НОВЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И СИСТЕМЫ (НИТИС 2019). Сборник научных статей XVI Международной научно-технической конференции, г. Пенза, 27-29 ноября 2019 г., с. 162-163
3. Kotlin programming language [Электронный ресурс] http://kotlinlang.org/
4. Ktor framework [Электронный ресурс] https://ktor.io/

ПРИЛОЖЕНИЕ А

Фрагменты исходного кода мобильного приложения

Файл WannaDressApplication.kt:

```
package application

import android.app.NotificationChannel
import android.app.NotificationManager
import android.content.Context
import android.content.IntentFilter
import android.os.Build
import androidx.localbroadcastmanager.content.LocalBroadcastManager
import androidx.multidex.MultiDex
import androidx.multidex.MultiDexApplication
import com.crashlytics.android.Crashlytics
import com.crashlytics.android.core.CrashlyticsCore
import com.maettaog.wannadress.BuildConfig
import com.maettaog.wannadress.R
import com.squareup.leakcanary.LeakCanary
import data.analytics.Analytics
import data.logging.DebugLogTree
import data.logging.ReleaseLogTree
import di.*
import io.fabric.sdk.android.Fabric
import org.koin.android.ext.android.startKoin
import presentation.notifications.NotificationDisplayer
import presentation.notifications.NotificationFactory
import timber.log.Timber
import utils.NetUtils
```

```
@Suppress("unused")
class WannaDressApplication : MultiDexApplication() {

    override fun attachBaseContext(base: Context?) {
        super.attachBaseContext(base)
        MultiDex.install(this)
    }

    override fun onCreate() {
        super.onCreate()

        Analytics.init(this)
        setupCrashReporting()
        setupTimber()
        setupLeakCanary()
        setupDependencyInjection()
        setupNotificationChannels()
        setupReceivers()

        NetUtils.init(this)
    }

    private fun setupCrashReporting() {
            val isDisabled = !(BuildConfig.BUILD_TYPE == "release" ||
BuildConfig.BUILD_TYPE == "debugCrashes")
        val crashlytics = Crashlytics.Builder()
            .core(
                CrashlyticsCore.Builder()
                    .disabled(isDisabled)
                    .build()
```

```
            )
            .build()
        Fabric.with(this, crashlytics)
    }

    private fun setupTimber() {
            val logTree = if (BuildConfig.DEBUG) DebugLogTree() else
ReleaseLogTree()
        Timber.plant(logTree)
    }

    private fun setupLeakCanary() {
        if (LeakCanary.isInAnalyzerProcess(this)) {
            return
        }
        LeakCanary.install(this)
    }

    private fun setupDependencyInjection() {
        val modules = listOf(
            commonModule, repositoriesModule, useCasesModule, apisModule,
            authModule, genderModule, mainModule
        )
        startKoin(this, modules)
    }

    private fun setupNotificationChannels() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val name = getString(R.string.daily_notification_channel_name)
            val importance = NotificationManager.IMPORTANCE_DEFAULT
```

```
                val channel = NotificationChannel(NotificationFactory.NOTIFICATION_CHANNEL_ID, name, importance)

                val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as? NotificationManager
                notificationManager?.createNotificationChannel(channel)
            }
        }

        private fun setupReceivers() {
            val action = getString(R.string.action_daily_notification)
            LocalBroadcastManager.getInstance(this).registerReceiver(NotificationDisplayer(), IntentFilter(action))
        }

    }
```

Файл WannaDressHttpClient.kt:

```
package data.network

import com.maettaog.wannadress.BuildConfig
import data.network.apiSpecs.UserApiSpec
import data.network.apiSpecs.WardrobeApiSpec
import data.network.apiSpecs.WeatherApiSpec
import data.network.apiSpecs.WhatToWearApiSpec
import okhttp3.Interceptor
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
```

```
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.gson.GsonConverterFactory
import utils.serverBaseUrl
import java.util.concurrent.TimeUnit

/**
 * Класс для конфигурации работы с http.
 */
class WannaDressHttpClient {

        private val client by lazy { configRetrofit(serverBaseUrl, BuildConfig.DEBUG) }
        val userApi: UserApiSpec by lazy { client.create(UserApiSpec::class.java) }
        val weatherApi: WeatherApiSpec by lazy { client.create(WeatherApiSpec::class.java) }
        val wardrobeApi: WardrobeApiSpec by lazy { client.create(WardrobeApiSpec::class.java) }
        val whatToWearApi: WhatToWearApiSpec by lazy { client.create(WhatToWearApiSpec::class.java) }

    private fun configRetrofit(baseUrl: String, isDebugging: Boolean): Retrofit {
        return Retrofit.Builder()
            .baseUrl(baseUrl)
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
            .addConverterFactory(GsonConverterFactory.create())
            .client(configHttpClient(isDebugging))
            .build()
    }
```

```
    private fun configHttpClient(isDebugging: Boolean): OkHttpClient {
        return OkHttpClient.Builder()
            .connectTimeout(TIMEOUT_SEC, TimeUnit.SECONDS)
            .readTimeout(TIMEOUT_READ_SEC, TimeUnit.SECONDS)
            .writeTimeout(TIMEOUT_WRITE_SEC, TimeUnit.SECONDS)
            .addInterceptor(configLoggingInterceptor(isDebugging))
            .build()
    }

    private fun configLoggingInterceptor(isDebugging: Boolean): Interceptor {
        return HttpLoggingInterceptor().apply {
            level = if (isDebugging) HttpLoggingInterceptor.Level.BODY else
HttpLoggingInterceptor.Level.NONE
        }
    }

    companion object {
        private const val TIMEOUT_SEC = 30L
        private const val TIMEOUT_READ_SEC = 30L
        private const val TIMEOUT_WRITE_SEC = 2 * 30L
    }
}
```

Файл UserApiSpec.kt:

```
package data.network.apiSpecs

import domain.models.responses.JWTValidationResponse
import domain.models.responses.TokenWrapper
import domain.models.responses.UserResponse
```

```
import io.reactivex.Completable
import io.reactivex.Single
import retrofit2.http.*

/**
 * Интерфейс для запросов, связанных с пользователем.
 *
 * @author Keyrillanskiy.
 */
interface UserApiSpec {
    /**
     * Метод для авторизации.
     *
      * @param token Токен авторизации, полученный из аккаунта (пока только google).
     *
     * @return JWT токен для авторизации на сервере.
     */
    @PUT("/api/auth")
    fun authorize(@Body token: TokenWrapper): Single<String>

    /**
     * Метод для получения информации о пользователе.
     *
     * @param jwt JWT токен авторизации.
     *
     * @return Информация о пользователе.
     */
    @GET("/api/user")
```

```
fun fetchUserData(@Header("Authorization") jwt: String): Single<UserResponse>

/**
 * Метод для изменения данных о пользователе.
 *
 * @param jwt JWT токен авторизации.
 * @param gender Пол пользователя.
 * @param temperatureConstant Температурная константа, которая влияет на алгоритм подбора одежды.
 */
@POST("/api/user")
fun updateUserData(
    @Header("Authorization") jwt: String,
    @Query("gender") gender: String,
    @Query("tconst") temperatureConstant: Int
): Completable

/**
 * Метод для удаления пользователя.
 *
 * @param jwt JWT токен авторизации.
 */
@DELETE("/api/user")
fun deleteUser(@Header("Authorization") jwt: String): Completable

/**
 * Метод для валидации токена авторизации.
 */
@GET("/api/validation")
```

```
    fun validateToken(@Header("Authorization") jwt: String): Single<JWTValidationResponse>

}
```

Файл WeatherApiSpec.kt:

```
package data.network.apiSpecs

import domain.models.responses.WeatherResponse
import io.reactivex.Single
import retrofit2.http.GET
import retrofit2.http.Query

/**
 * Интерфейс для запросов, связанных с погодой.
 *
 * @author Keyrillanskiy.
 */
interface WeatherApiSpec {

    /**
     * Запрос на сервер для получения текущей погоды и прогноза погоды.
     *
     * @param latitude Широта.
     * @param longitude Долгота.
     * @param language Язык погоды.
     *
     * @return Класс с текущей погодой и прогнозом погоды.
     */
```

```
    @GET("/api/weather")
    fun fetchWeather(
        @Query("latitude") latitude: Double,
        @Query("longitude") longitude: Double,
        @Query("lang") language: String
    ): Single<WeatherResponse>

}
```

Файл Preferences.kt:

```
package data.preferences

import android.content.Context
import com.google.gson.Gson
import domain.models.entities.User
import domain.models.responses.CurrentLocation

/**
 * Класс для работы с shared preferences.
 */
class Preferences(private val context: Context) {

                                private val prefs by lazy
{                                context.getSharedPreferences(PREFERENCES_NAME,
Context.MODE_PRIVATE) }
    private val gson by lazy { Gson() }

    fun isAuthorized(): Boolean = authToken != null

    var authToken: String?
```

```
        get() = prefs.getString(PREF_AUTH_TOKEN, null)
        set(value) {
            prefs.edit()
                .putString(PREF_AUTH_TOKEN, value)
                .apply()
        }

    private var userCache: User? = null
    var user: User?
        get() {
            fun loadUser(): User? {
                val userJson = prefs.getString(PREF_USER, null) ?: return null
                return gson.fromJson(userJson, User::class.java)
            }
            return userCache ?: loadUser()
        }
        set(value) {
            userCache = value

            val userJson = if (value == null) null else gson.toJson(value)
            prefs.edit()
                .putString(PREF_USER, userJson)
                .apply()
        }

                fun userOrException(): User = user ?: throw
IllegalArgumentException("User is not authorized")

    var isFirstLaunch: Boolean
        get() = prefs.getBoolean(PREF_IS_FIRST_LAUNCH, true)
```

```
        set(value) {
            prefs.edit()
                .putBoolean(PREF_IS_FIRST_LAUNCH, value)
                .apply()
        }

    var isDailyNotificationEnabled: Boolean
                                                    get() =
prefs.getBoolean(PREF_IS_DAILY_NOTIFICATIONS_ENABLED, false)
        set(value) {
            prefs.edit()
                .putBoolean(PREF_IS_DAILY_NOTIFICATIONS_ENABLED,
value)
                .apply()
        }

    var locationCache: CurrentLocation
        get() {
            val latitude = prefs.getFloat(PREF_LATITUDE, 0f).toDouble()
            val longitude = prefs.getFloat(PREF_LONGITUDE, 0f).toDouble()
            return CurrentLocation(latitude, longitude)
        }
        set(value) {
            prefs.edit()
                .putFloat(PREF_LATITUDE, value.latitude.toFloat())
                .putFloat(PREF_LONGITUDE, value.longitude.toFloat())
                .apply()
        }

    var locationLastModifiedTimestamp: Long
```

```
get() = prefs.getLong(PREF_LOCATION_LAST_MODIFIED_TIMESTAMP, 0)
set(value) {
prefs.edit()
.putLong(PREF_LOCATION_LAST_MODIFIED_TIMESTAMP, value)
.apply()
}

private companion object {
private const val PREFERENCES_NAME = "WannaDressPrefs"

private const val PREF_AUTH_TOKEN = "authToken"
private const val PREF_USER = "user"
private const val PREF_IS_FIRST_LAUNCH = "isFirstLaunch"
private const val PREF_IS_DAILY_NOTIFICATIONS_ENABLED = "isDailyNotificationsEnabled"
private const val PREF_LATITUDE = "latitude"
private const val PREF_LONGITUDE = "longitude"
private const val PREF_LOCATION_LAST_MODIFIED_TIMESTAMP = "locationLastModifiedTimestamp"
}

}
```

Файл LocationRepository.kt:

```
package data.repositories.interfaces

import com.google.android.gms.location.FusedLocationProviderClient
```

```
import domain.models.responses.CurrentLocation
import io.reactivex.Single

/**
 * Репозиторий для работы с местоположением.
 */
interface LocationRepository {

        fun fetchLocation(locationProvider: FusedLocationProviderClient): Single<CurrentLocation>

    fun cacheLocation(latitude: Double, longitude: Double)

}
```

Файл LocationRepositoryImpl.kt:

```
package data.repositories.implementations

import android.annotation.SuppressLint
import android.location.Location
import com.google.android.gms.location.FusedLocationProviderClient
import data.preferences.Preferences
import data.repositories.interfaces.LocationRepository
import domain.models.responses.CurrentLocation
import io.reactivex.Single
import java.util.concurrent.TimeUnit

/**
 * Имплементация репозитория для работы с местоположением.
 */
```

```
class LocationRepositoryImpl(private val preferences: Preferences) : LocationRepository {

    @SuppressLint("MissingPermission")
    override fun fetchLocation(locationProvider: FusedLocationProviderClient): Single<CurrentLocation> {
        return Single.create { single ->
            locationProvider.lastLocation
                .addOnSuccessListener { location: Location? ->
                    if (location != null) {
                        val currentLocation = CurrentLocation(location.latitude, location.longitude)
                        preferences.locationCache = currentLocation
                        preferences.locationLastModifiedTimestamp = System.currentTimeMillis()
                        single.onSuccess(currentLocation)
                    } else {
                        if (System.currentTimeMillis() - preferences.locationLastModifiedTimestamp > LOCATION_CACHE_LIFETIME) {
                            single.onError(IllegalArgumentException("Last location is null"))
                        } else {
                            single.onSuccess(preferences.locationCache)
                        }
                    }
                }
                .addOnFailureListener { error -> single.onError(error) }
        }
    }
```

```
        override fun cacheLocation(latitude: Double, longitude: Double) {
            val currentLocation = CurrentLocation(latitude, longitude)
            preferences.locationCache = currentLocation
                                    preferences.locationLastModifiedTimestamp =
System.currentTimeMillis()
        }

        private companion object {
                                private val LOCATION_CACHE_LIFETIME =
TimeUnit.HOURS.toMillis(4)
        }

    }
```

Файл RepositoriesModule.kt:

```
    package di

    import data.repositories.implementations.*
    import data.repositories.interfaces.*
    import org.koin.android.ext.koin.androidContext
    import org.koin.dsl.module.module

    /**
     * Модуль для управления зависимостями репозиториев.
     */
    val repositoriesModule = module {
        val resources = androidContext().resources

            single<UserRepository> { UserRepositoryImpl(httpClient = get(),
preferences = get()) }
```

```
        single<LocationRepository> { LocationRepositoryImpl(preferences = get()) }

        single<WeatherRepository> { WeatherRepositoryImpl(httpClient = get(), resources = resources) }

        single<WardrobeRepository> {
            WardrobeRepositoryImpl(
                httpClient = get(),
                resources = resources,
                preferences = get()
            )
        }

        single<WhatToWearRepository> {
            WhatToWearRepositoryImpl(
                httpClient = get(),
                resources = resources,
                preferences = get()
            )
        }
    }
```

Файл User.kt:

```
package domain.models.entities

import domain.models.values.Gender

/**
```

```
 * Модель пользователя
 *
 * @param id Строковое представление [java.util.UUID].
 * @param email Почта пользователя.
 * @param name Имя пользователя.
 * @param gender Пол пользователя.
 * @param avatarUrl Ссылка на аватар пользователя.
 * @param temperatureConstant Температурная константа, которая влияет на алгоритм подбора одежды.
 */
data class User(
    val id: String,
    val email: String,
    val name: String,
    val gender: Gender,
    val avatarUrl: String,
    val temperatureConstant: Int
)
```

Файл Gender.kt:

```
package domain.models.values

/**
 * Пол пользователя.
 */
enum class Gender(val stringValue: String) {
    MALE("male"),
    FEMALE("female")
}
```

```
fun String.toGender(): Gender {
    val genders = Gender.values()
    return genders.first { it.stringValue == this }
}
```

Файл Language.kt:

```
package domain.models.values

/**
 * Виды локализации в приложении для получения геолокации.
 */
enum class Language(val stringValue: String) {
    RUSSIAN("ru"), ENGLISH("en")
}

fun defineLanguage(language: String): Language {
        return Language.values().find { it.stringValue == language } ?: Language.ENGLISH
}
```

Файл ResponseWrapper.kt:

```
package presentation.common

/**
 * Класс для оборачивания данных, которые приходят из ViewModel.
 * Таким образом данные "приобретают 3 состояния": успешно (success),
 * ошибка (failure) и загрузка (loading).
*/

sealed class Response<out T> {
```

```
    companion object
}

data class Success<out T>(val value: T) : Response<T>()

data class Failure<out T>(val error: Throwable) : Response<T>()

class Loading<out T> : Response<T>()

fun <T> Response.Companion.success(value: T) = Success(value)

fun <T> Response.Companion.failure(error: Throwable) = Failure<T>(error)

fun <T> Response.Companion.loading() = Loading<T>()
```

Файл MainActivity.kt:

```
package presentation.screens.main

import android.content.Context
import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.os.Handler
import androidx.appcompat.app.AppCompatActivity
import com.google.android.gms.auth.api.signin.GoogleSignIn
import com.google.android.gms.auth.api.signin.GoogleSignInOptions
import com.maettaog.wannadress.R
import kotlinx.android.synthetic.main.activity_main.*
import org.koin.androidx.viewmodel.ext.android.viewModel
import presentation.screens.auth.AuthActivity
```

```
import
presentation.screens.main.MainPagerAdapter.Companion.CATEGORIES_FRAGME
NT_POSITION
import
presentation.screens.main.MainPagerAdapter.Companion.SETTINGS_FRAGMENT
_POSITION
import
presentation.screens.main.MainPagerAdapter.Companion.WEATHER_FRAGMENT
_POSITION
import presentation.screens.settings.SettingsInteractor
import presentation.screens.wardrobe.SuggestThingDialog
import presentation.screens.wardrobe.categories.WardrobeInteractor
import presentation.screens.weather.RequestLocationDialog
import presentation.screens.weather.WeatherInteractor
import presentation.share.ErrorDialog
import uk.co.deanwild.materialshowcaseview.MaterialShowcaseSequence
import uk.co.deanwild.materialshowcaseview.ShowcaseConfig
import utils.NetUtils
import utils.extensions.gone
import utils.extensions.toast
import utils.extensions.visible
import java.util.concurrent.TimeUnit


/**
 * Активити, которая содержит viewPager с экранами погоды, гардероба
и настроек.
*/
class MainActivity : AppCompatActivity(), WeatherInteractor,
    WardrobeInteractor, SettingsInteractor {
```

```
    private val viewModel by viewModel<MainViewModel>()
    private lateinit var viewInteractor: MainViewInteractor
    private val handler by lazy { Handler() }

    override fun onCreate(savedInstanceState: Bundle?) {
        setTheme(R.style.AppTheme_NoActionBar) // потому что по умолчанию стоит тема для splash
        super.onCreate(savedInstanceState)

        // routing
        when {
            !viewModel.isUserAuthorized() -> {
                AuthActivity.launch(this)
                finish()
            }
            else -> initMainScreen()
        }
    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        supportFragmentManager.fragments.forEach { it.onActivityResult(requestCode, resultCode, data) }
        super.onActivityResult(requestCode, resultCode, data)
    }

    override fun onBackPressed() {
        when (val currentItemPosition = mainViewPager.currentItem) {
            WEATHER_FRAGMENT_POSITION -> {
```

```
                if (viewModel.isBackButtonClickedOnceOnWeatherScreen) {
                    super.onBackPressed()
                    return
                }

                toast(R.string.double_press_to_exit)
                viewModel.isBackButtonClickedOnceOnWeatherScreen = true
                handler.postDelayed(
                        { viewModel.isBackButtonClickedOnceOnWeatherScreen =
false },
                    BACK_BUTTON_DOUBLE_PRESS_TIMEOUT
                )
            }
            CATEGORIES_FRAGMENT_POSITION -> {
                                    val currentChildFragmentManager =
viewInteractor.fragmentAdapter
                    .getFragment(CATEGORIES_FRAGMENT_POSITION)
                    .childFragmentManager
                if (currentChildFragmentManager.backStackEntryCount > 0) {
                    currentChildFragmentManager.popBackStack()
                } else {
                                        mainViewPager.currentItem =
WEATHER_FRAGMENT_POSITION
                    mainBottomNavigation.selectedItemId = R.id.weather
                }
            }
            SETTINGS_FRAGMENT_POSITION -> {
                                        mainViewPager.currentItem =
WEATHER_FRAGMENT_POSITION
                mainBottomNavigation.selectedItemId = R.id.weather
```

```
            }
                else -> throw IllegalArgumentException("invalid currentItem position: $currentItemPosition")
            }
        }

        override fun onShowLoading() {
            progressIndicator.visible()
        }

        override fun onHideLoading() {
            progressIndicator.gone()
        }

        override fun onInternetError() {
            val title = getString(R.string.internet_error_title)
            val message = getString(R.string.internet_connection_error_message)

            onShowErrorDialog(title, message)
        }

        override fun onLogOut() {
                                                    val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN).build()
            GoogleSignIn.getClient(this, gso).signOut()

            viewModel.deleteUserData()
            AuthActivity.launch(this)
            finish()
        }
```

```
        override fun onShowLocationPermissionReasonDialog(
            onPermitClick: (() -> Unit)?,
            onCancelClick: (() -> Unit)?
        ) {
            supportFragmentManager.findFragmentByTag(RequestLocationDialog.
TAG) as? RequestLocationDialog
                ?: RequestLocationDialog()
                    .also {
                        it.onPermitClick = { onPermitClick?.invoke() }
                        it.onCancelClick = { onCancelClick?.invoke() }
                        it.show(supportFragmentManager, RequestLocationDialog.TAG)
                    }
        }

        override fun onShowErrorDialog(
            title: String?,
            message: String?,
            onOkClick: (() -> Unit)?,
            onRetryClick: (() -> Unit)?
        ) {
            supportFragmentManager.findFragmentByTag(ErrorDialog.TAG) as?
ErrorDialog
                ?: ErrorDialog.newInstance(title, message)
                    .also { dialog ->
                        onOkClick?.let { dialog.onOkClick = { it.invoke() } }
                        onRetryClick?.let { dialog.onRetryClick = { it.invoke() } }
                        dialog.show(supportFragmentManager, ErrorDialog.TAG)
                    }
        }
```

```
        override fun onShowCase() {
            if (!viewModel.preferences.isFirstLaunch) {
                return
            }

            val config = ShowcaseConfig().apply {
                delay = 500
            }
            val circleItemOneString = getString(R.string.tutorial_circle_one)
            val circleItemTwoString = getString(R.string.tutorial_circle_two)
            val circleItemThreeString = getString(R.string.tutorial_circle_three)
            val gotItString = getString(R.string.got_it)
                                val sequence = MaterialShowcaseSequence(this,
SHOWCASE_ID).apply {
                setConfig(config)
                        addSequenceItem(mainBottomNavItem1, circleItemOneString,
gotItString)
                        addSequenceItem(mainBottomNavItem2, circleItemTwoString,
gotItString)
                       addSequenceItem(mainBottomNavItem3, circleItemThreeString,
gotItString)
            }
            sequence.start()

            viewModel.preferences.isFirstLaunch = false
        }

        override fun onNativeAdClick() {
```

```
                    val intent = Intent(Intent.ACTION_VIEW).apply { data =
Uri.parse("https://www.adidas.ru/") }
                startActivity(intent)
            }

            private fun initMainScreen() {
                setContentView(R.layout.activity_main)
                                        viewInteractor = MainViewInteractor(root,
supportFragmentManager).setup { /*empty*/ }
            }

            override fun onSettingsChanged() {
                NetUtils.withNetConnection(
                    onSuccess = { viewModel.uploadSettings() },
                    onError = {
                        viewModel.fetchCachedUserData()
                        showUploadSettingsInternetError()
                    }
                )
            }

            override fun onOpenSuggestThingDialog() {

                fun sendEmail(emailSubject: String, message: String) {
                    NetUtils.withNetConnection(
                        onSuccess = { viewModel.sendEmail(emailSubject, message) },
                        onError = { onSendError() }
                    )
                }
```

```
                val emailSubject = getString(R.string.suggest_thing_email_subject)
                 supportFragmentManager.findFragmentByTag(SuggestThingDialog.TAG) as? SuggestThingDialog
                    ?: SuggestThingDialog.newInstance()
                        .also { dialog ->
                                dialog.onSendClick = { message -> sendEmail(emailSubject, message) }
                                                                dialog.show(supportFragmentManager, SuggestThingDialog.TAG)
                        }
            }

            override fun onSendError() {
                val title = getString(R.string.unknown_error)
                val message = getString(R.string.internet_connection_error_message)

                onShowErrorDialog(title, message)
            }

            private fun showUploadSettingsInternetError() {
                val title = getString(R.string.upload_settings_error)
                val message = getString(R.string.internet_connection_error_message)

                                onShowErrorDialog(title, message, onRetryClick = { viewModel.uploadSettings() })
            }

            companion object {
                                private const val SHOWCASE_ID = "com.github.keyrillanskiy.main.showcase"
```

```
                private val BACK_BUTTON_DOUBLE_PRESS_TIMEOUT =
TimeUnit.SECONDS.toMillis(2)

        fun launch(context: Context) {
            context.startActivity(Intent(context, MainActivity::class.java))
        }
    }

}
```

ПРИЛОЖЕНИЕ В

Фрагменты исходного кода сервера

MainService

Файл Proxy.kt:

```
package com.maettaog

import io.ktor.application.call
import io.ktor.client.HttpClient
import io.ktor.client.engine.cio.CIO
import io.ktor.client.features.ClientRequestException
import io.ktor.client.features.json.JsonFeature
import io.ktor.client.features.json.serializer.KotlinxSerializer
import io.ktor.client.request.request
import io.ktor.client.request.url
import io.ktor.client.statement.HttpResponse
import io.ktor.client.statement.readBytes
import io.ktor.http.cio.Response
import io.ktor.request.httpMethod
import io.ktor.request.receiveOrNull
```

```
import io.ktor.request.uri
import io.ktor.response.respond
import io.ktor.routing.Route
import io.ktor.util.KtorExperimentalAPI
import io.ktor.util.filter
import kotlinx.serialization.UnstableDefault
import kotlinx.serialization.json.Json

/**
 * Http client, который выполняет обратное проксирование запросов.
 * Настроен так, чтобы не падать, если код ответа >= 300.
 */
@UnstableDefault
@KtorExperimentalAPI
private val httpClient by lazy {
    HttpClient(CIO) {
        expectSuccess = false
        install(JsonFeature) {
                serializer = KotlinxSerializer(json = Json.nonstrict) // nonstrict разрешает игнорировать неизвестные поля
        }
    }
}

/**
 * Выполняет обратное проксирование всех запросов на заданный [baseUrl].
 *
 * @param baseUrl Адрес для проксирования запроса.
 */
```

```
@KtorExperimentalAPI
@UnstableDefault
fun Route.reverseProxy(baseUrl: String) {
    handle {
        val response = httpClient.request<HttpResponse> {
            method = call.request.httpMethod
            url(baseUrl + call.request.uri)
             headers.appendAll(call.request.headers.filter { header, _ -> header != "Content-Length" })
              call.receiveOrNull<ByteArray>()?.let { receivedBody -> this.body = receivedBody }
        }
        call.respond(response.status, response.readBytes())
    }
}

/**
 * Общий путь к API приложения, который используется при роутинге прокси.
 */
const val apiRoute = "/api"
```

Файл WhatToWearProxy.kt:

```
package com.maettaog.proxies

import com.maettaog.apiRoute
import com.maettaog.reverseProxy
import io.ktor.application.Application
import io.ktor.routing.route
import io.ktor.routing.routing
```

```
import io.ktor.util.KtorExperimentalAPI
import kotlinx.serialization.UnstableDefault

@UnstableDefault
@KtorExperimentalAPI
fun Application.whatToWearModule() {
                                        val whatToWearUrl =
environment.config.propertyOrNull("ktor.deployment.whatToWearUrl")?.getString()
        ?: throw IllegalArgumentException("config field 'whatToWearUrl' is
null")

    routing {
        route(apiRoute) {
            route("/what_to_wear") { reverseProxy(whatToWearUrl) }
            route("/data_set") { reverseProxy(whatToWearUrl) }
        }
    }
}
```

Файл WeatherProxy.kt:

```
package com.maettaog.proxies

import com.maettaog.apiRoute
import com.maettaog.reverseProxy
import io.ktor.application.Application
import io.ktor.routing.route
import io.ktor.routing.routing
import io.ktor.util.KtorExperimentalAPI
import kotlinx.serialization.UnstableDefault
```

```
@UnstableDefault
@KtorExperimentalAPI
fun Application.weatherModule() {
                                                    val          weatherUrl          =
environment.config.propertyOrNull("ktor.deployment.weatherUrl")?.getString()
        ?: throw IllegalArgumentException("config field 'weatherUrl' is null")

    routing {
        route(apiRoute) {
            route("/weather") { reverseProxy(weatherUrl) }
        }
    }
}
```

WeatherService

Файл ApiView.kt:

```
package com.maettaog

import com.maettaog.models.BadRequestNoParamException
import com.maettaog.models.languageOrDefault
import io.ktor.application.Application
import io.ktor.application.call
import io.ktor.http.HttpStatusCode
import io.ktor.response.respond
import io.ktor.response.respondText
import io.ktor.routing.Route
import io.ktor.routing.get
import io.ktor.routing.route
import io.ktor.routing.routing
```

```
import io.ktor.util.KtorExperimentalAPI
import kotlinx.serialization.UnstableDefault

@UnstableDefault
@KtorExperimentalAPI
fun Application.apiModule() {
    routing {
        route("/api") {
            weatherEndpoint()
        }
    }
}

/**
 * Точка для получения погоды. Принимает широту, долготу и язык приложения пользователя.
 */
@UnstableDefault
@KtorExperimentalAPI
private fun Route.weatherEndpoint() {
    get("/weather") {
        try {
            val queryParams = call.request.queryParameters

            val latitudeParam = "latitude"
            val latitude = queryParams[latitudeParam]?.toDoubleOrNull()
                ?: throw BadRequestNoParamException(latitudeParam)

            val longitudeParam = "longitude"
            val longitude = queryParams[longitudeParam]?.toDoubleOrNull()
```

```
                ?: throw BadRequestNoParamException(longitudeParam)

            val languageParam = "lang"
            val language = queryParams[languageParam].languageOrDefault()

            call.respond(weatherByCriteria(latitude, longitude, language))
        } catch (e: BadRequestNoParamException) {
            call.respondText(
                "Param '${e.missingParam}' is not specified or invalid",
                status = HttpStatusCode.BadRequest
            )
        }
    }
}
```

Файл WeatherController.kt:

```
package com.maettaog

import com.maettaog.models.Language
import com.maettaog.models.Weather
import com.maettaog.models.WeatherForClient
import com.maettaog.models.YandexWeatherResponse
import io.ktor.client.HttpClient
import io.ktor.client.engine.cio.CIO
import io.ktor.client.features.json.JsonFeature
import io.ktor.client.features.json.serializer.KotlinxSerializer
import io.ktor.client.request.get
import io.ktor.client.request.header
import io.ktor.client.request.parameter
import io.ktor.client.request.url
```

```
import io.ktor.util.KtorExperimentalAPI
import kotlinx.serialization.UnstableDefault
import kotlinx.serialization.json.Json

/**
 * HttpClient, который используется для получения погоды.
 */
@UnstableDefault
@KtorExperimentalAPI
private val httpClient by lazy {
    HttpClient(CIO) {
        install(JsonFeature) {
                serializer = KotlinxSerializer(json = Json.nonstrict) // nonstrict разрешает игнорировать неизвестные поля
        }
    }
}

/**
 * Функция для получения погоды.
 */
@UnstableDefault
@KtorExperimentalAPI
suspend fun weatherByCriteria(latitude: Double, longitude: Double, language: Language): Weather {
    return httpClient.get<YandexWeatherResponse> {
        url("https://api.weather.yandex.ru/v1/forecast")
        header("X-Yandex-API-Key", yandexWeatherApiKey)
        url {
            parameter("lat", latitude)
```

```
            parameter("lon", longitude)
            parameter("lang", language.yandexWeatherValue)
            parameter("limit", 1) // this maybe doesn't work now
            parameter("hours", true)
            parameter("extra", true) // for extra info about precipitation
        }
    }.toWeather(language)
}
```

ImageService

Файл ApiView.kt:

```
package com.maettaog

import com.maettaog.controllers.categoryNamesToIconUrlsMap
import com.maettaog.controllers.generateWhatToWearImage
import com.maettaog.controllers.thingsNamesToIconUrlsMap
import com.maettaog.models.BadRequestNoParamException
import com.maettaog.models.NotFoundException
import com.maettaog.models.parseGenderOrNull
import io.ktor.application.Application
import io.ktor.application.application
import io.ktor.application.call
import io.ktor.http.ContentType
import io.ktor.http.HttpStatusCode
import io.ktor.http.content.resources
import io.ktor.http.content.static
import io.ktor.response.respond
import io.ktor.response.respondOutputStream
import io.ktor.response.respondText
```

```
import io.ktor.routing.*
import io.ktor.util.KtorExperimentalAPI
import kotlinx.serialization.UnstableDefault
import javax.imageio.ImageIO

const val staticRoute = "static"
const val resourcesRoute = "resources"
const val categoriesRoute = "category_icons"

@UnstableDefault
@KtorExperimentalAPI
fun Application.apiModule() {
    routing {
        static("/$staticRoute") {
            resources(staticRoute)
        }

        route("/api") {
            categoriesEndpoint()
            thingsEndpoint()
            whatToWearImageEndpoint()
        }
    }
}

/**
 * Точка для получения ссылок на изображения категорий (для отображения в гардеробе).
 *
 * Принимает пол и массив названий изображений.
```

```
     * Возвращает json, где ключ - название изображения, а значение - ссылка на изображение.
     */
    @KtorExperimentalAPI
    private fun Route.categoriesEndpoint() {
        get("/$categoriesRoute") {
                                                    val serverBaseUrl = application.environment.config.propertyOrNull("ktor.application.url")?.getString()
                ?: throw IllegalArgumentException("config field 'url' is null")

            call.respond(categoryNamesToIconUrlsMap(serverBaseUrl))
        }
    }

    /**
     * Точка для получения ссылок на изображения вещей в масштабе иконок (для отображения в гардеробе, например).
     *
     * Принимает пол и массив названий изображений.
     * Возвращает json, где ключ - название изображения, а значение - ссылка на изображение.
     */
    @UnstableDefault
    @KtorExperimentalAPI
    private fun Route.thingsEndpoint() {
        post("/thing_icons") {
            try {
                val genderParam = "gender"
                                                        val gender = call.request.queryParameters[genderParam]?.parseGenderOrNull()
```

```
                ?: throw BadRequestNoParamException(genderParam)

            val namesParam = "names"
                val thingNames = call.parameters.getAll(namesParam) ?: throw BadRequestNoParamException(namesParam)

                                                    val serverBaseUrl = application.environment.config.propertyOrNull("ktor.application.url")?.getString()
                ?: throw IllegalArgumentException("config field 'url' is null")

                    call.respond(thingsNamesToIconUrlsMap(gender, thingNames, serverBaseUrl))
            } catch (e: BadRequestNoParamException) {
                call.respondText(
                    "Param '${e.missingParam}' is not specified or invalid",
                    status = HttpStatusCode.BadRequest
                )
            }
        }
    }

    /**
     * Точка для получения сгенерированного изображения одетого человечка.
     *
     * Принимает пол и массив названий изображений.
     * Возвращает сгенерированное изображение.
     */
    private fun Route.whatToWearImageEndpoint() {
        get("/what_to_wear") {
```

```
            try {
                val queryParams = call.request.queryParameters

                val genderParam = "gender"
                    val gender = queryParams[genderParam]?.parseGenderOrNull() ?: throw BadRequestNoParamException(genderParam)

                val namesParam = "names"
                        val thingNames = queryParams.getAll(namesParam) ?: throw BadRequestNoParamException(namesParam)

                            val bufferedImage = generateWhatToWearImage(gender, thingNames)
                call.respondOutputStream(
                    contentType = ContentType.Image.PNG,
                    producer = { ImageIO.write(bufferedImage, "png", this) }
                )
            } catch (e: BadRequestNoParamException) {
                call.respondText(
                    "Param '${e.missingParam}' is not specified or invalid",
                    status = HttpStatusCode.BadRequest
                )
            } catch (e: NotFoundException) {
                call.respondText(
                    "Object with name '${e.missingParam}' is not found",
                    status = HttpStatusCode.NotFound
                )
            }
        }
    }
```

Файл ThingsController.kt:

```
package com.maettaog.controllers

import com.maettaog.models.Gender
import com.maettaog.models.NotFoundException
import com.maettaog.resourcesRoute
import com.maettaog.staticRoute
import kotlinx.io.IOException
import java.awt.image.BufferedImage
import java.io.File
import javax.imageio.ImageIO

private const val thingsRoute = "things"

/**
 * Получает все требуемые ссылки на изображения вещей и возвращает map,
 * в котором ключ - название вещи, а значение - ссылка на изображение.
 *
 * @param gender Пол.
 * @param thingNames Имена вещей.
 * @param serverBaseUrl Адрес сервера, который будет использован в ссылке на изображение.
 */
fun thingsNamesToIconUrlsMap(gender: Gender, thingNames: List<String>, serverBaseUrl: String): Map<String, String> {
    val iconSizeRoute = "iconSize"
    val thingsFolder = File("$resourcesRoute/$staticRoute/$thingsRoute/$iconSizeRoute/${gender.stringValue}")
```

```
                val allFileNames = thingsFolder.list() ?: throw IllegalArgumentException("things folder is empty")

            val requestedImageUrls = allFileNames
                .mapNotNull { fileName ->
                    (thingNames.firstOrNull { it == fileName.substringBefore(".png") }
                        ?: return@mapNotNull null) to fileName
                }
                .map { (thingName, fileName) ->
                        thingName to "$serverBaseUrl/$staticRoute/$thingsRoute/$iconSizeRoute/${gender.stringValue}/$fileName"
                }

            return requestedImageUrls.toMap()
        }

        /**
         * Генерирует изображение человечка в одежде. Вещи отрисовываются одна на другую по порядку в списке [thingNames].
         *
         * @param gender Пол.
         * @param thingNames Имена вещей для отрисовки на человечке.
         */
        fun generateWhatToWearImage(gender: Gender, thingNames: List<String>): BufferedImage {
            val personsRoute = "persons"
            val personSizeRoute = "personSize"
                val personFile = File("$resourcesRoute/$staticRoute/$personsRoute/${gender.stringValue}.png")
            val personImage = ImageIO.read(personFile)
```

```
            val thingsFolder = File("$resourcesRoute/$staticRoute/$thingsRoute/
$personSizeRoute/${gender.stringValue}")
        for (thingName in thingNames) {
            try {
                                val thingImage = ImageIO.read(File(thingsFolder,
"$thingName.png"))
                personImage.graphics.drawImage(thingImage, 0, 0, null)
            } catch (e: IOException) {
                throw NotFoundException(thingName)
            }
        }

        return personImage
    }
```

ПРИЛОЖЕНИЕ С

Презентация