ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данная дипломная работа посвящена разработке интерактивной карты с различными достопримечательностями БГУ, как научно-исследовательскими, так и учебными. Эта карта содержит в себе информацию о местоположении каждой из достопримечательностей. Такая карта позволит пользователю быстро и удобно понять, где расположена та или иная достопримечательность, а также получить некоторую информацию о ней.

Разработка данной карты безусловно является достаточно актуальной и значимой. Её можно было бы использовать во многих сферах жизни: научно-исследовательской, исторической, туристической и многих других. Более того, проект с разработкой данной карты будет несложно модифицировать и улучшать в будущем, тем самым добавляя в него намного больше возможностей и функционала.

Данный проект отвечает за мобильное приложение с этой интерактивной картой под операционную систему Android. Основная задача мобильного приложения – получить данные о достопримечательностях с веб-сервера и расположить их на интерфейсе пользователя, а также добавить функционал и различные способы взаимодействия с элементами карты. Таким образом, пользователи смогут пользоваться функционалом карты прямо с мобильного устройства, что может быть очень востребовано, например, в туристической сфере. Если говорить про пользователей операционной системы iOS, то в рамках проекта bsu-map для них предусмотрено отдельное мобильное приложение другим разработчиком.

Для создания мобильного приложения с данной интерактивной картой понадобились некоторые технологии Android-разработки, такие как: Kotlin Coroutines, Room Database, Retrofit2, Okhttp3, Koin, WorkManager и многие другие. Однако основополагающей технологией является Mapbox, которая позволяет работать с картами, метками и даже маршрутами. Комбинация вышеуказанных технологий позволила разработать мобильное приложение с интерактивной картой, создание которого планировалось в данной дипломной работе.

ГЛАВА 1
ИЗУЧЕНИЕ ТЕХНОЛОГИЙ РАЗРАБОТКИ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ ПОД ОПЕРАЦИОННУЮ СИСТЕМУ ANDROID

1.1 Технологии android-разработки

1.1.1 WorkManager

WorkManager – это API, который упрощает планирование надежных асинхронных задач, которые, как ожидается, будут выполняться даже при выходе из приложения или перезапуске устройства. WorkManager API является подходящей и рекомендованной заменой для всех предыдущих API фонового планирования Android, включая FirebaseJobDispatcher, GcmNetworkManager и Job Scheduler . WorkManager включает в себя функции своих предшественников в современном последовательном API, который работает до уровня API 14, но при этом учитывает время автономной работы.

WorkManager использует базовую службу диспетчеризации заданий на основе следующих критериев (см. рис. 1.1):

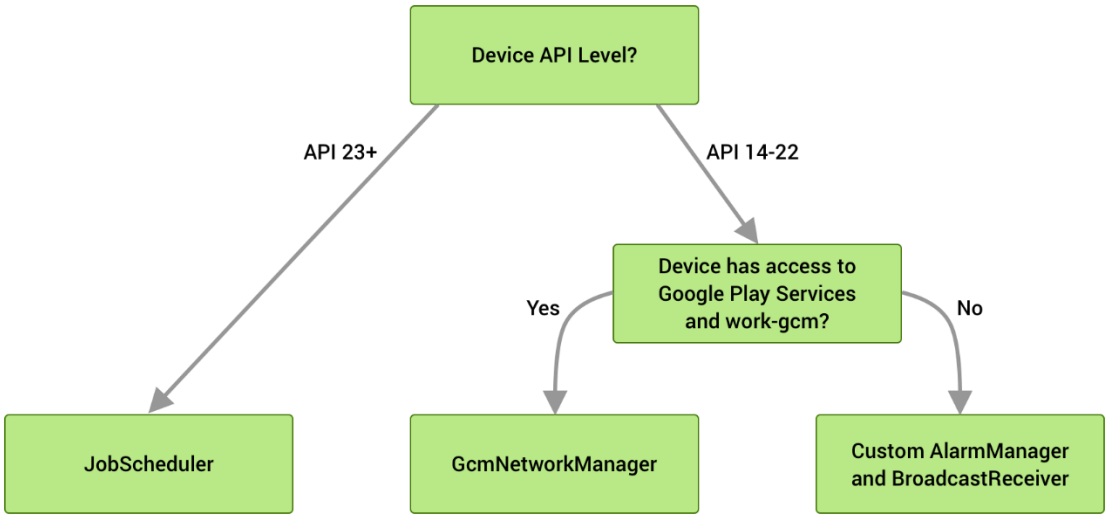

Рисунок 1.1 - Базовая служба диспетчеризации заданий WokManager

Для работы с WorkManager требуется декларативно определить оптимальные условия для работы, используя рабочие ограничения. Например, запускать, только когда устройство подключено к сети Wi-Fi, когда устройство находится в режиме ожидания, или когда на нем достаточно места для хранения.

WorkManager позволяет планировать работу, чтобы выполнить её одноразово или несколько раз с использованием гибких окон планирования. Работа также может быть помечена тегами и названа, что позволяет планировать уникальную заменяемую работу, а также отслеживать или отменять группы работ вместе. Запланированная работа хранится во внутренне управляемой базе

данных SQLite, и WorkManager заботится о том, чтобы эта работа сохранялась и переносилась при перезагрузке устройства. Кроме того, WorkManager придерживается функций энергосбережения и передовых методов, таких как режим Doze, поэтому об этом можно не беспокоиться.

Иногда работа не получается. WorkManager предлагает гибкие политики повторных попыток, включая настраиваемую политику экспоненциальной отсрочки.

Для сложной связанной работы можно объединить отдельные рабочие задачи в цепочку с помощью понятного, естественного интерфейса, который позволяет контролировать, какие части выполняются последовательно, а какие – параллельно.

Для каждой рабочей задачи можно определить входные и выходные данные для соответствующей работы. При объединении работы в цепочку WorkManager автоматически передает выходные данные от одной рабочей задачи к другой.

WorkManager интегрируется с RxJava и Coroutines и обеспечивает гибкость в собственных асинхронных интерфейсах.

WorkManager предназначен для работы, которая требует надежности, даже если пользователь переходит за пределы экрана, приложение закрывается или устройство перезагружается. Например:

- Отправка журналов или аналитики в серверные службы
- Периодическая синхронизация данных приложения с сервером

WorkManager не предназначен для фоновой работы в процессе, которая может быть безопасно прекращена, если процесс приложения прекращается, или для работы, требующей немедленного выполнения. Существует руководство по фоновой обработке, которое позволяет узнать, какое решение соответствует тем или иным потребностям.

Данная технология может пригодиться при разработке интерактивной карты. Например, если пользователь мобильного приложения желает, чтобы данные о новых достопримечательностях автоматически поступали на его мобильное устройство, он может включить соответствующую функцию, которая будет загружать эти данные на фоне, даже тогда, когда пользователь не использует приложение.

1.1.2 Dependency Injection. Koin

Dependency Injection – это метод программирования, который делает класс независимым от его зависимостей. Это стало возможным благодаря разделению использования объекта и его создания. Многие разработчики Android знакомы с

основанными на Java фреймворками внедрения зависимостей, такими как Dagger и Guice.

Однако некоторые фреймворки полностью написаны на Kotlin. Эти фреймворки включают Koin и Kodein. Далее будет отмечаться то, как управлять зависимостями в Android с помощью новой инфраструктуры внедрения зависимостей – Koin.

Фреймворк внедрения зависимостей помогает создавать и управлять зависимостями. Как упоминалось ранее, существует множество основанных на Java фреймворков для внедрения зависимостей Android. Однако с ростом внедрения Kotlin на Android растет спрос на библиотеки, полностью написанные на Kotlin.

Koin – это инфраструктура внедрения зависимостей, которая соответствует этой потребности. Это легкий фреймворк, простой в освоении и не содержащий большого количества шаблонного кода. Эту структуру можно эффективно использовать для управления зависимостями в приложениях под операционную систему Android.

Благодаря мощи языка Kotlin, Koin предоставляет DSL, чтобы описать приложение, вместо того, чтобы аннотировать его или генерировать для него код. Благодаря Kotlin DSL, Koin предлагает интеллектуальный функциональный API для подготовки инъекции зависимостей.

Koin предлагает несколько ключевых слов, позволяющих описать элементы приложения Koin:

- Application DSL, чтобы описать конфигурацию контейнера Koin
- Модуль DSL, чтобы описать компоненты, которые необходимо ввести.

Экземпляр KoinApplication представляет собой конфигурацию экземпляра контейнера Koin. Это позволяет настроить ведение журнала, загрузку свойств и модули.

Чтобы построить новый KoinApplication, используются следующие функции:

- koinApplication {…} – создать KoinApplication конфигурацию контейнера
- startKoin {…} – создать KoinApplication конфигурацию контейнера и зарегистрировать ее в GlobalContext, чтобы разрешить использование GlobalContext API.

Чтобы настроить KoinApplication экземпляр, можно использовать любую из следующих функций:

- logger() – описывает, какой уровень и реализацию Logger использовать (по умолчанию используется EmptyLogger)

- modules() – устанавливает список модулей Koin для загрузки в контейнер

- properties() – загружает свойства HashMap в контейнер Koin

- fileProperties() – загружает свойства из заданного файла в контейнер Koin

- environmentProperties() – загружает свойства из окружения ОС в контейнер Koin

- createEagerInstances() – создаёт нетерпеливые экземпляры (отдельные определения отмечены как createdAtStart)

Можно описать конфигурацию контейнера Koin двумя способами: koinApplication или startKoin функцией.

- koinApplication описывает экземпляр контейнера Koin

- startKoin описывает экземпляр контейнера Koin и регистрирует его в Koin GlobalContext

Зарегистрировав конфигурацию контейнера в GlobalContext, глобальный API может использовать ее напрямую. Любой KoinComponent относится к Koin экземпляру. По умолчанию мы используем экземпляр из GlobalContext.

Модуль Koin – это «пространство» для сбора определений Koin. Он объявляется с помощью module функции.

Объявление одноэлементного компонента означает, что контейнер Koin будет хранить уникальный экземпляр объявленного компонента. Так, мы можем использовать single функцию в модуле, чтобы объявить синглтон.

Single, factory и scoped – это ключевые слова, которые объявляют компоненты через лямбда-выражения. Данное лямбдя-выражение описывает способ создания компонента. Обычно мы создаем экземпляры компонентов через их конструкторов, но можно также использовать любое выражение. Пример объявления компоненты:

single { Class constructor // Kotlin expression }

Тип результата лямбда-выражения – это основной тип компонента.

Koin может быть очень удобен при разработке мобильного приложения, так как он позволяет хранить все зависимости проекта в одном файле. Благодаря

этой технологии можно избавиться от ряда потенциальных ошибок при компиляции и сборке, что является большим преимуществом.

1.1.3 Retrofit2. Okhttp3

Многие сайты имеют собственные API для удобного доступа к своим данным. На данный момент самый распространённый вариант – это JSON. Также могут встречаться данные в виде XML и других форматов. Библиотека Retrofit упрощает взаимодействие с REST API сайта, беря на себя часть рутинной работы. Авторами библиотеки Retrofit являются разработчики из компании "Square", которые написали множество полезных библиотек, например, Picasso, Okhttp, Otto.

Данной библиотекой удобно пользоваться для запроса к различным веб-сервисам с командами GET, POST, PUT, DELETE. Данная технология также позволяет работать в асинхронном режиме, что избавляет от лишнего кода. Стоит отметить, что, при подключении к проекту библиотеки Retrofit2, библиотека Okhttp подключается автоматически, что также очень удобно.

Ниже представлен пример кода, позволяющий создать объект Retrofit:

```
Retrofit retrofit = Retrofit.Builder()

.baseUrl("https://your.api.url/v2/");

.addConverterFactory(ProtoConverterFactory.create())

.addConverterFactory(GsonConverterFactory.create())
.build();
```

В объекте Retrofit мы можем указать ссылку на API веб-сервера, с которым мы работаем. Помимо этого, мы можем указать необходимый нам конвертер данных из существующих, либо использовать собственный конвертер, реализовав интерфейс на основе абстрактного класса Converter.Factory.

Ниже представлен список аннотацией, которые активно используются Retrofit:

- @GET() – GET-запрос для базового адреса. Также можно указать параметры в скобках

- @POST() – POST-запрос для базового адреса. Также можно указать параметры в скобках

- @Path – Переменная для замещения конечной точки, например, username подставится в {username} в адресе конечной точки

- @Query – Задаёт имя ключа запроса со значением параметра

- @Body – Используется в POST-вызовах (из Java-объекта в JSON-строку)

- @Header – Задаёт заголовок со значением параметра

- @Headers – Задаёт все заголовки вместе

- @Multipart – Используется при загрузке файлов или изображений

- @FormUrlEncoded – Используется при использовании пары "имя/значение" в POST-запросах

- @FieldMap – Используется при использовании пары "имя/значение" в POST-запросах

- @Url – Для поддержки динамических адресов

Так как мобильное приложение вынуждено забирать данные о достопримечательностях с сервера, то ему понадобятся технологии, с помощью которых станет возможным получить доступ к API веб-сервера. Таким образом, библиотеки Retrofit2 и Okhttp3 будут очень востребованы в приложении.

1.1.4 Room Database

Приложения, которые обрабатывают нетривиальные объемы структурированных данных, могут значительно выиграть от локального сохранения этих данных. Наиболее распространенный вариант использования - кэширование соответствующих фрагментов данных, чтобы, когда устройство не может получить доступ к сети, пользователь все еще мог просматривать этот контент, находясь в автономном режиме.

Библиотека Room обеспечивает уровень абстракции над SQLite, чтобы обеспечить свободный доступ к базе данных, используя всю мощь SQLite. В частности, Room предоставляет следующие преимущества:

- Проверка SQL-запросов во время компиляции.

- Удобные аннотации, которые сводят к минимуму повторяющийся и подверженный ошибкам шаблонный код.

- Оптимизированные пути миграции базы данных.

Благодаря этим преимуществам можно использовать Room вместо непосредственного использования API SQLite.

Room состоит из трех основных компонентов:

1) Класс базы данных, который содержит саму базу данных и служит основной точкой доступа для базового подключения к постоянным данным приложения.

2) Сущности данных, представляющие таблицы в базе данных приложения.

3) Объекты доступа к данным (DAO), которые предоставляют методы, которые приложение может использовать для запроса, обновления, вставки и удаления данных в базе данных.

Класс базы данных предоставляет приложению экземпляры DAO, связанные с этой базой данных. В свою очередь, приложение может использовать DAO для извлечения данных из базы данных как экземпляров связанных объектов сущностей данных.

Приложение также может использовать определенные объекты данных для обновления строк из соответствующих таблиц или для создания новых строк для вставки. Существует тесная взаимосвязь между компонентами Room (см. рис. 1.2):

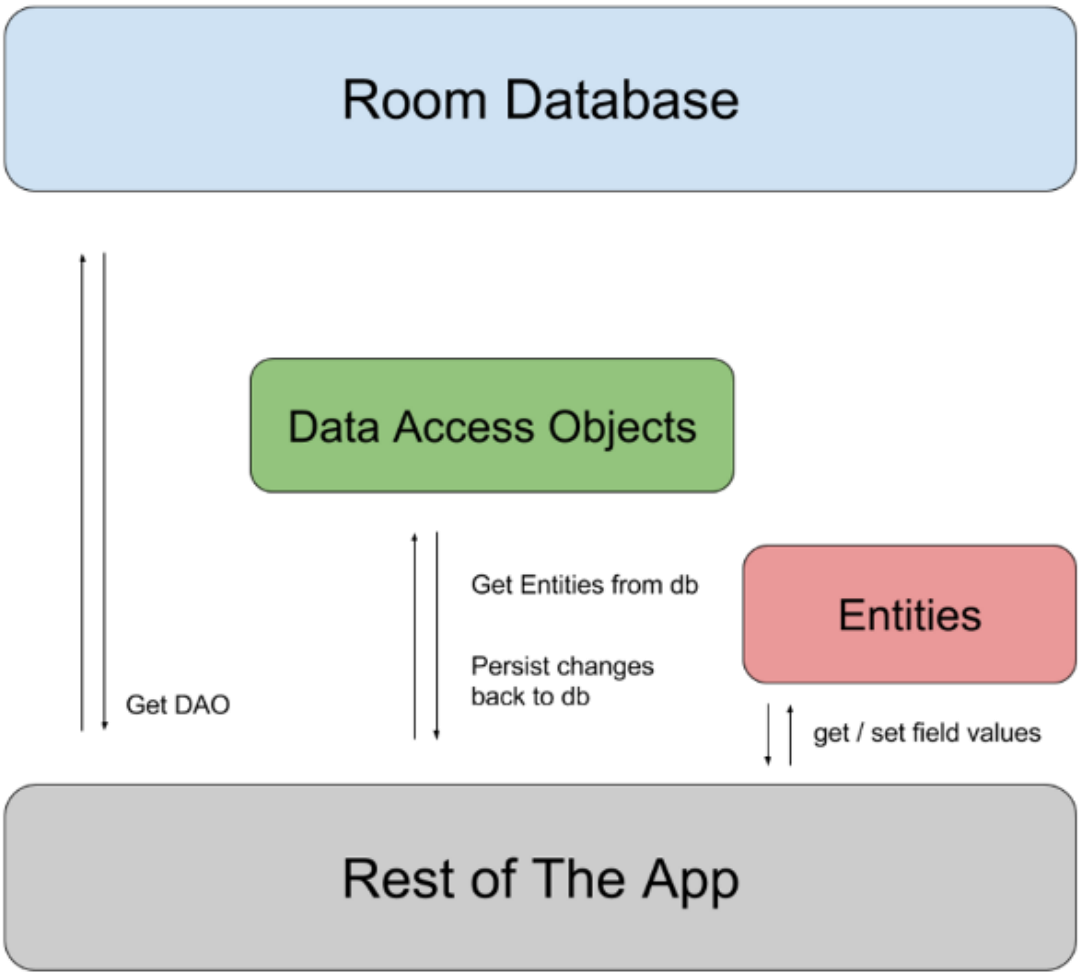

Рисунок 1.2 – Взаимосвязь между компонентами Room

Следующий код определяет AppDatabaseкласс для хранения базы данных:

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
abstract fun userDao(): UserDao
}
```

AppDatabase определяет конфигурацию базы данных и служит основной точкой доступа приложения к сохраненным данным. Класс базы данных должен удовлетворять следующим условиям:

- Класс должен быть аннотирован @Database аннотацией, которая включает в себя entities массив, в котором перечислены все объекты данных, связанные с базой данных.

- Класс должен быть расширяемым абстрактным классом RoomDatabase.

- Для каждого класса DAO, связанного с базой данных, класс базы данных должен определять абстрактный метод, который не имеет аргументов и возвращает экземпляр класса DAO.

Если приложение выполняется в рамках одного процесса, при создании экземпляра AppDatabase объекта нужно следовать шаблону проектирования singleton. Каждый RoomDatabase экземпляр стоит довольно дорого, и редко требуется доступ к нескольким экземплярам в рамках одного процесса. Если же приложение выполняется в нескольких процессах, его следует заключить в enableMultiInstanceInvalidation() вызов построителя базы данных. Таким образом, если есть экземпляр AppDatabase в каждом процессе, то можно сделать недействительным файл общей базы данных в одном процессе, и это признание автоматически распространится на экземпляры AppDatabasев других процессов.

После того, как определён объект данных, DAO и объект базы данных, можно использовать следующий код для создания экземпляра базы данных:

```
val db = Room.databaseBuilder(

applicationContext,

AppDatabase::class.java, "database-name"

).build()
```

Затем можно использовать абстрактные методы из AppDatabase, чтобы получить экземпляр DAO. Например, можно использовать методы из экземпляра DAO для взаимодействия с базой данных:

```
val userDao = db.userDao()

val users: List<User> = userDao.getAll()
```

Безусловно, значение создания локальной базы данных для приложения с интерактивной картой очень велико. В случае, если у пользователя приложения не будет доступа к Интернету, он всё равно сможет получить информацию о местоположениях достопримечательностей, если те были добавлены в локальную базу данных ранее.

1.1.5 Kotlin Coroutines

Coroutines – это технология языка Kotlin, позволяющая работать с многопоточными приложениями. Данная технология позволяет писать асинхронный неблокирующий код последовательным образом.

В Kotlin Coroutines есть специальная функция, которую мы можем объявить с помощью ключевого слова suspend. Приостановка функций может приостановить выполнение сопрограммы, что означает, что она будет ждать, пока приостанавливающие функции не возобновятся. Примером работы сопрограммы может служить следующий код:

```
CoroutineScope(Dispatchers.Main + Job()).launch {
val user = fetchUser() // A suspending function running in the I/O thread.
updateUser(user) // Updates UI in the main thread.
}
private suspend fun fetchUser(): User = withContext(Dispatchers.IO) {
// Fetches the data from server and returns user data.
}
```

В данном коде можно выделить основополагающие компоненты сопрограммы (см. рис. 1.3):

Рисунок 1.3 – Компоненты Kotlin Coroutines

Конструкция CoroutineScope определяет область действия новых сопрограмм. Каждый построитель сопрограмм является расширением

CoroutineScope и наследует его coroutineContext для автоматического распространения как элементов контекста, так и отмены.

MainScope же способен создавать основную область для компонентов пользовательского интерфейса. Он работает в основном потоке с SupervisorJob(), что означает, что сбой одного из его дочерних заданий не повлияет на другие.

Существует также GlobalScope. Это область, которая не привязана к какой-либо работе. Она используется для запуска сопрограмм верхнего уровня, которые работают в течение всего времени жизни приложения и не отменяются преждевременно.

Сопрограмма всегда выполняется в некотором контексте, представленном значением типа CoroutineContext. CoroutineContext представляет собой набор элементов для определения политики потоковой передачи, обработчика исключений, управления временем жизни сопрограммы. Мы можем использовать оператор плюса для объединения элементов CoroutineContext.

Есть три наиболее важных контекста Coroutine – Dispatchers, CoroutineExceptionHandler и Job.

Dispatchers определяют, какой поток запускает сопрограмму. Сопрограмма может переключать диспетчеров в любое время с помощью withContext(). Подразделяют несколько разновидностей Dispatchers:

1) Dispatchers.Default – использует общий фоновый пул потоков. По умолчанию максимальное количество потоков, используемых этим диспетчером, равно количеству ядер ЦП, но не менее двух

2) Dispatchers.IO – разделяет потоки с Dispatchers.Default, но количество потоков ограничено kotlinx.coroutines.io.parallelism. По умолчанию используется ограничение в 64 потока или количество ядер (в зависимости от того, что больше)

3) Dispatchers.Main – соответствует основному потоку Android

4) Dispatchers.Unconfined – диспетчер сопрограмм, не ограниченный каким-либо конкретным потоком. Сопрограмма сначала выполняется в текущем потоке и позволяет ей возобновить работу в любом потоке, который используется соответствующей функцией приостановки

CoroutineExceptionHandler отвечает за обработку не перехваченных исключений. Как правило, не перехваченные исключения могут быть только результатом сопрограмм, созданных с помощью запуска строителя. Сопрограмма, созданная с использованием async, всегда перехватывает все свои исключения и представляет их в результирующем объекте Deferred.

Job – управляет временем жизни сопрограммы. У задания есть следующие состояния: isActive, isCompleted, isCancelled. Можно использовать метод Job.isActive, чтобы узнать текущее состояние задания (см. рис. 1.4):

State	isActive	isCompleted	isCancelled
New (optional initial state)	`false`	`false`	`false`
Active (default initial state)	`true`	`false`	`false`
Completing (transient state)	`true`	`false`	`false`
Cancelling (transient state)	`false`	`false`	`true`
Cancelled (final state)	`false`	`true`	`true`
Completed (final state)	`false`	`true`	`false`

Рисунок 1.4 – Состояния Job

Можно также привести схему потока смены состояний (см. рис. 1.5):

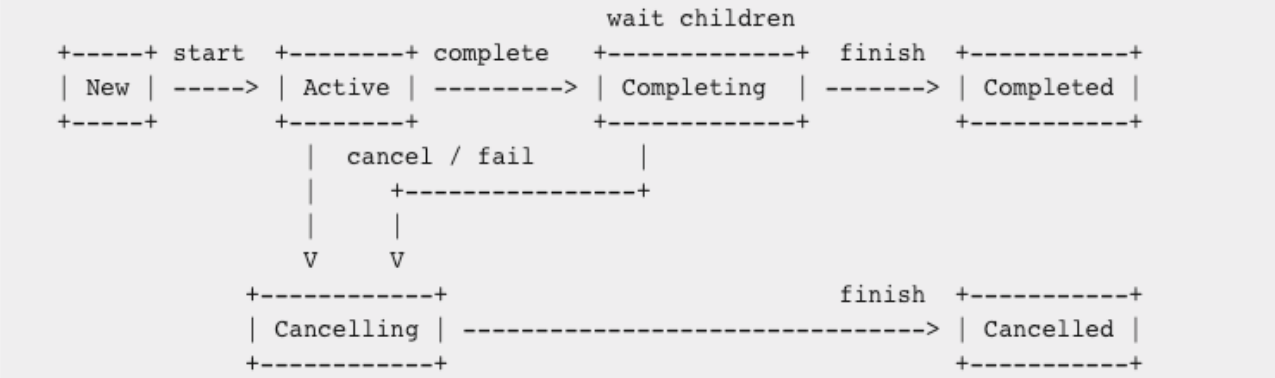

Рисунок 1.5 – Поток смены состояний Job

Задание активно, пока работает сопрограмма.

Если задание не выполнено, не считая исключения, оно отменяется. Задание можно отменить в любое время с помощью функции отмены, которая заставляет его немедленно перейти в состояние отмены.

Задание отменяется, когда завершается выполнение своей работы.

Родительское задание в состоянии завершения или отмены ожидает завершения всех своих дочерних задач перед завершением. Стоит обратить внимание, что состояние завершения является внутренним для задания. Для стороннего наблюдателя завершающее задание все еще активно, в то время как внутренне оно ожидает своих дочерних элементов.

Безусловно, значимость Kotlin Coroutines очень велика и для мобильного приложения с интерактивной картой. Например, для отслеживания состояния приложения необходимо запустить отдельный поток, который будет регистрировать результаты работы тех или иных функций, после чего выводить для пользователя на интерфейс соответствующее сообщение. Отдельные потоки

также могут понадобиться за тем, чтобы обновлять локальную базу данных с информацией о достопримечательностях в фоновом режиме, пока пользователь использует приложение.

1.2 Технологии разработки интерактивных карт

1.2.1 Разновидности API для работы с картами

В наши дни существует множество различных технологий, позволяющих разработчикам по всему миру разрабатывать свои продукты, связанные с использованием интерактивных карт. Эти технологии упрощают и ускоряют разработку различных элементов интерактивной карты, позволяя разработчикам затрачивать меньше времени на написание исходного кода, а также предоставляя широкий выбор шаблонов для тех или иных элементов карты.

На данный момент среди всевозможных API, связанных с разработкой интерактивных карт, можно выделить следующие:

- Google Maps
- Bing Maps
- Mapbox
- Foursquare
- Fencer
- Mapillary
- Yandex
- YAddress
- OpenStreetMap

Вышеуказанные технологии получили высокий уровень развития и являются достаточно удобными для использования в различных проектах. Однако, проект, на котором завязана данная дипломная работа, имеет достаточно ограниченный бюджет, из-за чего не может себе позволить работу с какой-либо из вышеперечисленных технологий. По этой причине было принято решение рассмотреть другие, более дешёвые варианты технологий. В качестве

кандидатов были выбраны технологии OpenStreetMap и, разработанная на её основе, Mapbox.

Перед реализацией проекта технологии OpenStreetMap и Mapbox были тщательно изучены и проанализированы. Были учтены следующие факторы:

- время, необходимое для изучения API
- возможности работы с объектами на карте
- производительность и возможность кэширования
- наличие карт различной местности

Таким образом, была выдвинута необходимость в исследовании двух вышеупомянутых технологий.

1.2.2 OpenStreetMap

OpenStreetMap – это карта всего мира, которую может редактировать каждый. Данная карта создается практически с чистого листа по GPS-трекам и распространяется под свободной лицензией.

Лицензия OpenStreetMap позволяет свободно (или почти свободно) получать доступ ко всем её растровым картам и всем лежащим в их основе пространственным данным. Этот проект направлен на поощрение нового и интересного использования этих данных.

Взглянуть на карту OSM достаточно просто. Создание карты ещё не завершено, но можно заметить, что некоторые места уже прорисованы довольно детально. Отдельные люди и целые коммерческие компании постоянно помогают в улучшении и детализации карт. Данный проект также не является исключением и способствует улучшению OSM.

OSM – это не просто программный проект. Для того, чтобы создать карты, технология отрывается от экранов компьютеров пользователей и исследует города и деревни, делая новую съёмку местности. Это делается коллективными усилиями сообщества (на данный процесс затрачивается очень много времени). Для этого используется wiki-подобное программное обеспечение, которое позволяет совместно редактировать карты. Это значит, что карты все время становятся больше и лучше. Если у пользователя есть GPS-приемник, он может сделать свой вклад в проект, загрузив любые записанные им треки. Кроме того, пользователь может рисовать при помощи спутниковых снимков Bing. Любой разработчик может начать редактировать карту прямо сейчас при помощи онлайн-редактора Potlatch или скачав программы JOSM или Merkaartor.

Множество программных разработок продвигают этот проект в разных направлениях. Как уже упоминалось выше, создаются различные инструменты

для редактирования карт. На самом деле OpenStreetMap живёт за счет энтузиазма и программного обеспечения с открытым исходным кодом, начиная от интерфейса карты и вплоть до лежащего в основе протокола доступа к данным (интерфейс веб-сервиса для чтения и записи данных карты). Существует много возможностей для дополнительных проектов, работающих с данными OSM и использующими их, но данной технологии также необходима помощь в исправлении ошибок и добавлении возможностей в её функционал.

Если сравнивать OSM и достаточно популярные Google Maps, то можно выделить достаточно весомое преимущество у первой технологии – открытый исходный код и свободный доступ к растровым картам. Например, при использовании Google Maps приходится выделять намного больше экономических ресурсов на проект, который их использует. В этом плане OSM выигрывает у Google Maps. Это – основная причина, почему данная технология была выбрана в качестве одного из кандидатов для разработки мобильного приложения с интерактивной картой.

Любые изменения в базе OSM соотносятся с так называемыми наборами изменений (changesets), которые могут быть отменены целиком или частично.

В OSM хранится вся история изменений, потому любой объект или набор объектов может быть возвращён к предыдущему состоянию на любую дату, а текущее состояние данных может быть сверено с предыдущим как по свойствам, так и по геометрии.

Существуют сервисы, анализирующие изменения за определённый последний период времени, показывающие характер правок (внесены новые объекты, отредактированы или удалены имеющиеся). Примерами таких сервисов могут быть: OSMCha, WHODIDIT, AChaVi.

Проект OSM существует благодаря участникам, знакомым с местностью, где они живут, а потому вандализм и саботаж, как правило, могут быть замечены, проанализированы и нейтрализованы в сравнительно короткий срок.

Также позволяют выявлять намеренное или случайное нарушение целостности данных так называемые "валидаторы" – сервисы, которые анализируют логическую целостность данных OSM (топологию административных границ и т.п.).

В функционал мобильного приложения с интерактивной картой достопримечательностей БГУ входит расстановка объектов на карте, размещение различной информации об этих объектах, а также, в перспективе, возможность построения маршрутов между этими объектами. Помимо того, в проекте было бы перспективно иметь возможность организовать работу со слоями на карте. Например, отдельный слой карты для исторических достопримечательностей и отдельный слой карты для современных достопримечательностей. Всё это можно реализовать при помощи OpenStreetMap.

1.2.3 Mapbox

Технология Mapbox возникла благодаря команде веб-разработчиков, которые решали следующую задачу: обеспечение возможности динамически рисовать карту внутри веб-браузера вместо того, чтобы загружать статические фрагменты карты, отображаемые на сервере. Они хотели встроить динамические, интерактивные, настраиваемые карты на веб-страницы и мобильные устройства, и они объединили векторные плитки и технологию 3D-рендеринга, чтобы создать данное решение.

С созданием Mapbox GL и спецификации стиля Mapbox с открытым исходным кодом открылся мир приложений динамического картографирования, и Mapbox стала компанией, занимающейся созданием инструментов, которые предоставляют эти возможности в руки разработчиков.

На сегодняшний день Mapbox поддерживает карты и службы определения местоположения для широкого спектра веб-приложений, мобильных, автомобильных и игровых приложений. В этом разделе описываются основные технологии, обеспечивающие работу служб Mapbox.

Накопленные картографические данные являются основой для многих служб определения местоположения. Конвейеры обработки данных, принадлежащие Mapbox, принимают новые данные с мобильных датчиков, отзывов водителей, камер с компьютерным зрением и аэрофотоснимков в специальные конвейеры обработки данных и объединяют эти данные с открытыми и собственными источниками, чтобы картографические данные всегда были актуальными. Mapbox также предлагает премиальные продукты для данных о границах, трафике и движении.

Данные о трафике, предоставляемые Mapbox, используют одни и те же конвейеры обработки для обновления профилей трафика для миллиардов сегментов дорог по всему миру каждые несколько минут, обеспечивая мощные навигационные возможности, такие как маршрутизация с учетом трафика и интуитивно понятные пошаговые инструкции.

Технология технического зрения использует эффективные нейронные сети для обработки изображений на уровне дорог непосредственно на мобильных или встроенных устройствах, превращая смартфоны и видеорегистраторы в живые датчики, интерпретирующие дорожные условия в режиме реального времени.

Имеющиеся инструменты для разработчиков включают в себя мощные API для служб карт, поиска, навигации, зрения и учетных записей. Разработчики данной технологии также предоставляют SDK, чтобы сделать эти услуги доступными для веб-разработчиков, разработчиков мобильных устройств, игр и встроенных устройств. Заготовленные инструменты для разработчиков включают Mapbox GL JS, специальный JavaScript SDK для веб-разработчиков; Mapbox Studio, бесплатный редактор стилей карты с визуальным предварительным просмотром в реальном времени; и Mapbox Atlas, а также локальное приложение для клиентов с ограниченными требованиями к сети. Полный список инструментов имеется в документации Mapbox.

Технология полагается на векторные листы для хранения и обслуживания большей части специальных картографических данных. Формат векторных листов компактен и предназначен для кэширования, масштабирования и быстрого обслуживания картографических данных. Векторные листы содержат геометрию и метаданные, которые можно отобразить на карте.

Имеющиеся графические библиотеки Mapbox GL и Mapbox GL Native сообщают веб-устройствам и мобильным устройствам, как рисовать карты в виде визуальной графики. Mapbox GL извлекает геопространственные данные из векторных листов и инструкции по стилю из документа стиля и помогает клиенту рисовать 2D- и 3D-карты Mapbox в виде динамической визуальной графики с помощью OpenGL.

Mapbox GL опирается на два документа спецификации, которые определяют стандарты, позволяющие Mapbox GL правильно инструктировать клиентов:

Спецификация векторных плиток Mapbox с открытым исходным кодом описывает, как геометрия и атрибуты в геопространственных данных должны храниться и кодироваться в векторных плитках.

Спецификация стилей Mapbox с открытым исходным кодом описывает, как следует написать стиль карты, чтобы сообщить Mapbox GL, какие данные рисовать, в каком порядке их рисовать, а также какие цвета, уровни непрозрачности и другие свойства применять при отображении данных. Редактор стилей Mapbox Studio — это визуальный инструмент для создания документа стиля, соответствующего этой спецификации.

Технологии Mapbox поддерживают множество возможностей определения местоположения, которые объединяются в нижеперечисленные основные службы.

Сервис Maps поддерживает данные карты, обслуживаемые через основные наборы фрагментов, и все специальные возможности для рендеринга этих данных в 2D и 3D, включая Mapbox GL и связанные спецификации, Mapbox Tiling Service, Mapbox Studio и Mapbox Atlas. Этот сервис включает в себя все API Карт и SDK, принадлежащие Mapbox.

Особая служба поиска привязана ко всему, что создает пользователь. Этот сервис включает в себя специальный API геокодирования, расширенные данные POI и специально разработанный SDK для поиска, который помогает разработчикам реализовывать такие функции, как поиск по категориям, нечеткий поиск, опережающее ввод текста, автозаполнение, историю пользователей и избранное.

Служба навигации использует особые механизмы маршрутизации для предоставления возможностей маршрутизации с учетом трафика и интуитивно понятных пошаговых указаний. Этот сервис включает в себя API-интерфейсы Directions, Map Matching, Isochrone, Optimization и Matrix, а также SDK Navigation, который помогает разработчикам добавлять пошаговую навигацию в мобильные приложения.

Существует также служба Vision, которая поддерживает возможности обработки изображений на основе ИИ в SDK Mapbox Vision и разрабатывает сам SDK.

На основе вышеупомянутой информации можно сделать вывод, что услуги Mapbox отличаются высокой функциональной совместимостью и что сервисные группы внимательно прислушиваются к отзывам клиентов, чтобы создавать продукты, отвечающие потребностям разработчиков. Данная технология, как и OpenStreetMap, хорошо подходит для разработки проекта с интерактивной картой.

1.2.4 Сравнительная оценка OpenStreetMap и Mapbox

Итак, выше была проведена тщательная оценка технологий OpenStreetMap и Mapbox, однако для проекта требовалось выбрать одну из данных технологий. Таким образом, во время разработки проекта возникла необходимость проанализировать достоинства и недостатки каждой из технологий, после чего провести сравнительный анализ и принять решение о выборе той или иной технологии.

Подчеркнём достоинства технологии OpenStreetMap:

- API OpenStreetMap является бесплатным
- Карта с открытым исходным кодом

Во-первых, данная технология является бесплатной. Это означает, что её можно использовать в любом проекте, включая коммерческие. Это безусловно является достоинством, так как исчезает необходимость в трате бюджета проекта.

Во-вторых, данная технология постоянно развивается. Существует множество разработчиков, активно занимающихся картированием, что позволяет обеспечить постоянный рост базы данных с картами.

Подчеркнём недостатки технологии OpenStreetMap:

- Требует создания дополнительных сервисов
- Ограниченное количество запросов

Стоит отметить, что API OpenStreetMap был разработан с целью обновления и редактирования карт, из-за чего имеет не самую широкую функциональность. Вследствие этого разработчикам приходится либо создавать необходимую инфраструктуру самостоятельно, либо использовать существующие комплексные решения, которые основаны на OSM. Это нельзя не назвать недостатком.

В качестве второго недостатка данной технологии можно отметить то, что из-за характера проекта чрезмерный обмен данными через API не рекомендуется, из-за чего пользователи технологии могут получить блокировку без предварительного уведомления об этом в случае, если они запрашивают слишком много данных.

После тщательного анализа и оценки возможностей технологии OpenStreetMap были проведены аналогичные исследования и для технологии Mapbox.

Подчеркнём достоинства технологии Mapbox:

- Индивидуальность и гибкость
- Быстрое время загрузки и отличная производительность
- Автономный режим в API
- Подход с открытым исходным кодом
- Стандартизированная обработка данных

Как можно заметить, Mapbox имеет ряд достоинств. Первым из них является то, что в случаях, когда работа над проектом предполагает особый дизайн и разработку, пользователь получает свободу действий с помощью Mapbox Studio и API Mapbox. С помощью данных инструментов пользователь имеет возможность добавлять объекты и выбирать слои для отображения, а также изменять цвета и шрифты, если это необходимо.

В качестве второго достоинства Mapbox можно выделить достаточно эффективную работу с большими массивами данных. С помощью архитектуры набора тайлов и оптимизации Mapbox GL JS разработчики могут ожидать от своих проектов быстрой и плавной загрузки интегрированных карт, особенно со сложными структурами данных.

Третье достоинство данной технологии заключается в том, что обеспечивается полная поддержка всех функций в офлайн-картах, а также в том, что количество плиток, которые можно загрузить, не ограничено. Это может быть полезно в тех случаях, когда подключение для передачи и обработки данных не может быть установлено или же оно не оправдано. Это особенно часто возникает во время международных поездок, походов или в тех случаях, когда требуется максимально оптимизировать скорость загрузки карты.

Ещё одним достоинством является то, что, как и в технологии OpenStreetMap, Mapbox публикует свой код и призывает сообщество разработчиков проверять и улучшить его. Стоит также отметить наличие библиотеки Mapbox GL Native, которая позволяет вставлять настраиваемые

интерактивные карты в нативные приложения под операционные системы iOS и Android.

В качестве последнего достоинства, которое можно отметить, выступает работа с достаточно строгими внутренними правилами управления данными Mapbox. Данная работа требует некоторого изучения и привыкания, однако в долгосрочной перспективе это окупается, вследствие чего разработчики могут эффективно разрабатывать интерактивные карты.

Подчеркнём недостатки технологии Mapbox:

- Плохое покрытие карты в некоторых регионах
- Существует кривая обучения Mapbox API

Как было упомянуто выше, технология Mapbox основывается на коллективном картографировании, а технология OpenStreetMap является его основным источником данных. И хотя за последнее время проект OSM получил достаточно широкое развитие, его охват в некоторых странах мира, таких, как, например, Китай и Индия, все ещё нуждается в некоторой доработке. Другими словами, одним из недостатков является не самое лучшее покрытие карт в некоторых точках мира.

В качестве второго недостатка данной технологии стоит отметить то, что для изучения её основ, а именно потока данных и стандартизации, требуется достаточно много времени, из-за чего разработчики могут столкнуться с некоторыми трудностями.

Ниже приведена таблица, которая наглядно сравнивает достоинства и недостатки обеих технологий (см. табл. 1):

Таблица 1 – Сравнительная оценка технологий OpenStreetMap и Mapbox

Технология	Преимущества	Недостатки
OpenStreetMap	• API OpenStreetMap является бесплатным • Карта с открытым исходным кодом	• Требует создания дополнительных сервисов • Ограниченное количество запросов
MapBox	• Индивидуальность и гибкость • Быстрое время загрузки и отличная производительность	• Плохое покрытие карты в некоторых регионах • Достаточно долгий процесс

	• Автономный режим в API • Подход с открытым исходным кодом • Стандартизированная обработка данных	изучения Mapbox API разработчиками

Таким образом, была успешно проведена оценка обеих технологий. Обе технологии предоставляют разработчикам статические и динамические карты, а также точные и подробные географические и локальные данные в различных выбираемых представлениях, возможность добавления маркеров, а также полигонов и изображений. Помимо этого, имеются направления в реальном времени, которые учитывают трафик, а также оптимизацию маршрута и оценку времени прибытия. Есть также некоторая информация о местах и достопримечательностях, а помимо того ещё и автоматический прогноз, и коррекция, предложения на основе текущего местоположения.

В плане настройки OSM значительно уступает технологии Mapbox. У второй есть ряд преимуществ, которых нету у первой. Таким образом, учитывая вышеупомянутые достоинства и недостатки обеих технологий, было принято решение продолжить разработку проекта с интерактивной картой с помощью технологии Mapbox.

ГЛАВА 2
РЕАЛИЗАЦИЯ ИЗУЧЕННЫХ ТЕХНОЛОГИЙ РАЗРАБОТКИ В ПРОЕКТЕ С ИНТЕРАКТИВНОЙ КАРТОЙ

2.1 Настройка среды и создание проекта

Прежде, чем приступать к любому проекту, крайне важно выбрать и настроить среду для работы с этим проектом. Так как в рамках данного проекта речь идёт о мобильном приложении под операционную систему Android, в качестве среды разработки была выбрана Android Studio.

Android Studio – это официальная объединенная среда разработки (IDE) для разработки приложений для Android. Данный инструмент основан на IntelliJ IDEA, интегрированной среде разработки программного обеспечения на языке Java, и включает в себя инструменты разработки кода и средства разработки.

Для поддержки разработки приложений в среде Android Studio используется система на основе Gradle, а также эмулятор, сборка шаблонов кода и интеграция с Github. Каждый проект в Android Studio имеет одну или несколько модальностей с исходным кодом и файловыми ресурсами. Эти модальности включают модули приложений Android, библиотеки и модули Google App Engine.

Android Studio использует функцию Instant Push для отправки изменений кода и ресурсов в работающее приложение. Редактор кода помогает разработчику в написании кода и предлагает завершение кода, преломление и анализ. Приложения, созданные в Android Studio, компилируются в формате APK для отправки в Google Play Store.

Android Studio предлагает широкий спектр в плане выбора шаблона для реализации проекта. Среди доступных есть: *No Activity*, *Basic Activity*, *Bottom Navigation Activity*, *Empty Compose Activity*, *Empty Activity*, *Fullscreen Activity*, *Google Maps Activity*, *Login Activity*, *Responsive Activity*, *Settings Activity*, Scrolling Activity, *Tabbed Activity* и другие.

После настройки среды был создан проект с названием *BSU Interactive Map*. Данный проект был создан на основе шаблона пустого Activity, после чего постепенно развивался. Один из этапов создания проекта представлен на рисунке ниже (см. рис. 2.1):

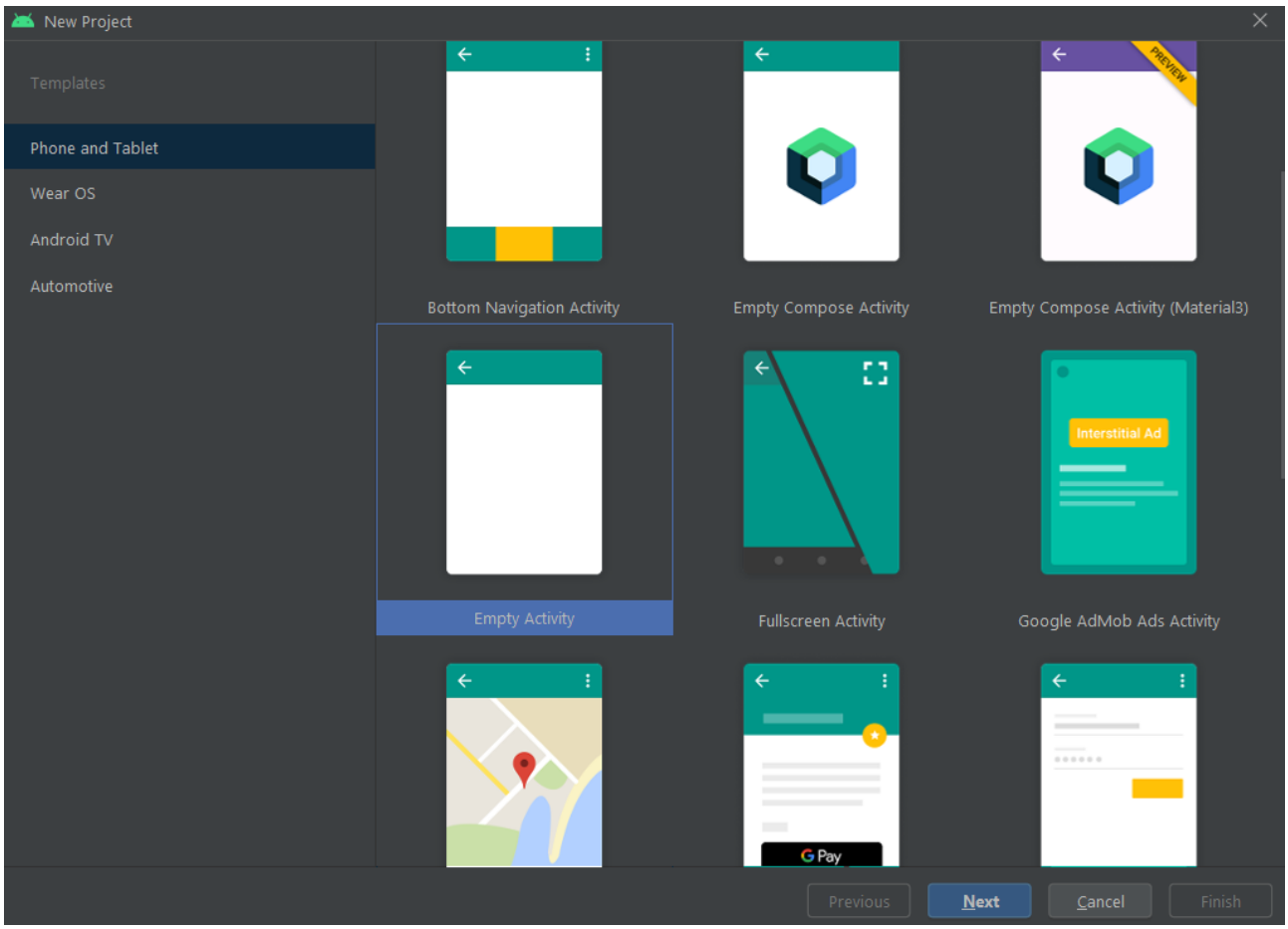

Рисунок 2.1 – Один из этапов создания проекта в Android Studio

После того, как проект был создан, было принято решение создать специальные модули для хранения классов и интерфейсов в нужных частях структуры проекта.

2.2 Создание проектных модулей

Проект с интерактивной картой включал в себя следующие модули:

- app
- data
- domain
- ui

Необходимость в создании модулей объясняется тем, что при такой политике построения структуры проекта разработчик имеет возможность отделить одну логику от другой. Например, создание модулей позволяет отделить логику сущностей базы данных и репозитория с данными от логики пользовательского интерфейса. При грамотном построении модулей можно вносить изменения в одни модули, при этом не затрагивая другие модули. Более того, хранение классов и интерфейсов в отдельных модулях обеспечивает хранение зависимостей в каждом из модулей, что положительно сказывается на эффективности компиляции кода.

Структура модулей проекта представлена на рисунке ниже (см. рис. 2.2):

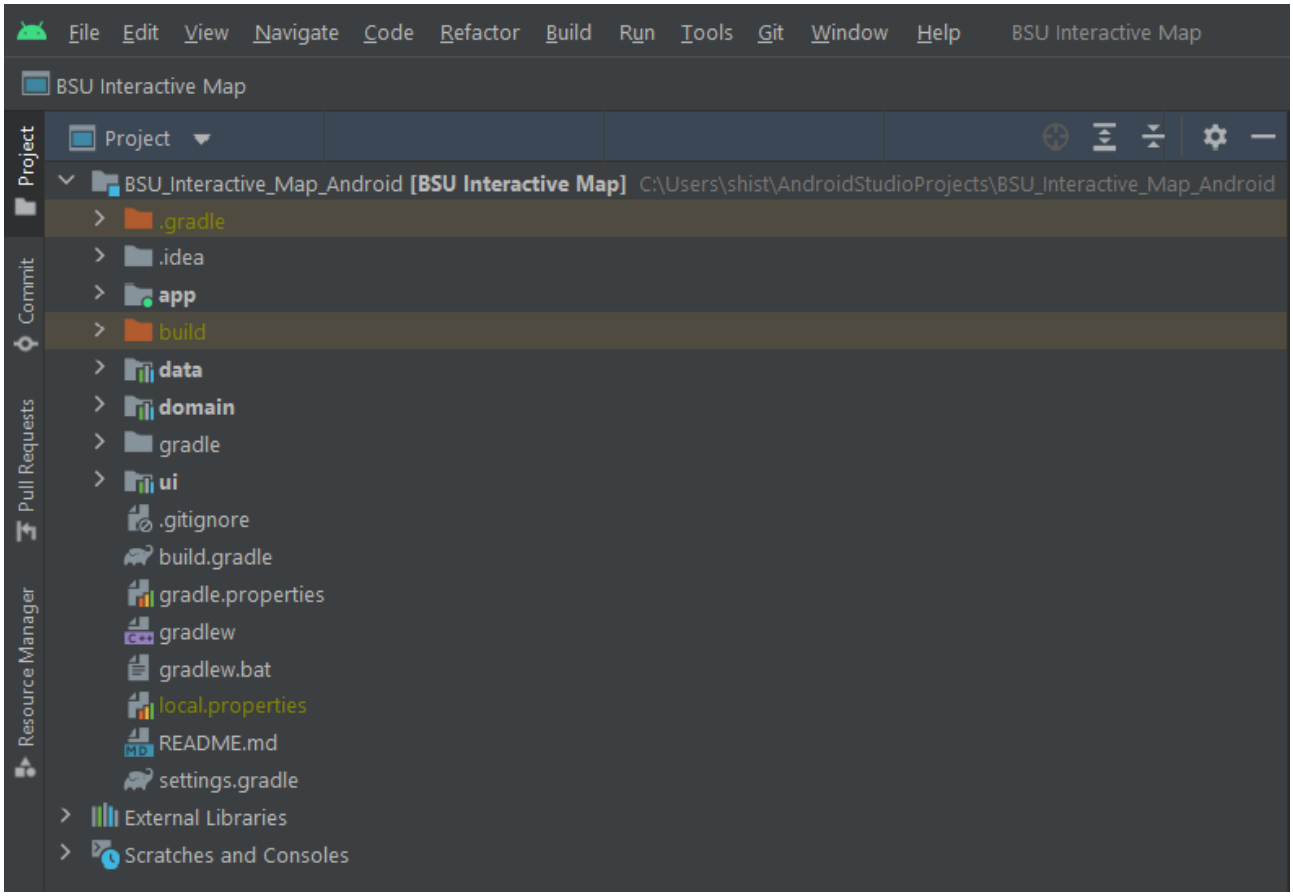

Рисунок 2.2 – Структура модулей проекта

Модуль *app* был разработан для того, чтобы поместить туда логику работы приложения в фоновом режиме. В данном модуле выполняются те инструкции, которые приложение выполняет в те моменты, когда пользователь пользуется другими приложениями или вовсе держит своё мобильное устройство неактивным.

В частности, в данном модуле находится класс *WorkManagerApplication*, который отвечает за то, чтобы обновлять данные приложения периодически. Об этом классе будет подробнее описано далее.

2.3 Реализация обновления данных в фоновом режиме

Одной из задач проекта было обеспечение возможности загрузки новых данных с веб-сервера в те моменты, когда приложение не запущено, а также когда мобильное устройство пользователя вовсе неактивно.

Для выполнения данной задачи был создан класс *WorkManagerApplication*. Этот класс содержит в себе инструкции, основанные на технологии WorkManager, которые позволяет приложению обновлять свои данные каждый раз в 8 часов утра, в случае, если пользователю доступна беспроводная сеть Wi-Fi и мобильное устройство пользователя находится на подзарядке.

Данные инструкции были реализованы с помощью следующего исходного кода:

```
val currentDate = Calendar.getInstance()

val dueDate = Calendar.getInstance()

dueDate[Calendar.HOUR_OF_DAY] = 8

dueDate[Calendar.MINUTE] = 0

dueDate[Calendar.SECOND] = 0

if(dueDate.before(currentDate)) {

dueDate.add(Calendar.DAY_OF_MONTH, 1)

}

val timeDiff = dueDate.timeInMillis - currentDate.timeInMillis

val constraints = Constraints.Builder()

        .setRequiredNetworkType(NetworkType.UNMETERED) // Only if Wi-Fi
is connected

        .setRequiresCharging(true) // Only if device is on recharging

        .build()

val saveRequest =
```

```
PeriodicWorkRequestBuilder<UploadWorker>(24, TimeUnit.HOURS)
        .setInitialDelay(timeDiff, TimeUnit.MILLISECONDS)
        .setConstraints(constraints)
        .build()
WorkManager.getInstance(applicationContext).enqueue(saveRequest)
```

Сначала создается объект календаря на основе текущего дня, после чего в его параметры (час, минута, секунда) вставляются значения, соответствующие нужному времени.

В случае, если полученное время уже является прошедшим (например, если у пользователя полдень, а приложение выставит значение на 8 утра), к объекту прибавляется один день. Таким образом, обновление данных произойдёт завтра в 8 утра.

Переменная *timeDiff* фиксирует в себе разницу между завтрашним днём (8 утра) и текущим днем с текущим времени, и передает полученное значение в качестве параметра в специальный запрос. Данная разница фактически измеряется в милисекундах.

Переменная *constraints* хранит в себе необходимые настройки для запроса. Например, в данном проекте было принято решение выставить данную настройку только в тех случаях, когда пользователь подключен к Wi-Fi и при этом его мобильное устройство находится на подзарядке. Так как процесс загрузки данных с сервера может быть достаточно затратным по трафику и электроэнергии, данная настройка показывается себя достаточно надёжно, ведь не каждый пользователь готов качать данные с мобильного интернета или с почти разряженного мобильного устройства. В общем, опцию с подобной настройкой в перспективе можно реализовать в качестве одной из настроек приложения.

Наконец, создается запрос в объекте *saveRequest* с периодичностью в одни стуки, который потом отправляется в очередь WorkManager.

2.4 Внедрение зависимостей

Одной из задач повышения производительности и ускорения компиляции кода было внедрение зависимостей в проект с помощью технологии Koin. Данная технология позволяет хранить объявления основных переменных и классов (например, ViewModel) в отдельных модулях, что предоставляет следующие преимущества:

- уменьшение вероятности ошибок
- ускорение компиляции
- улучшение структуры проекта

Всего в проекте было создано 2 Koin-модуля: uiModule и dataModule. Первый модуль отвечает за внедрение зависимостей для модуля пользовательского интерфейса, а второй – за внедрение зависимостей для модуля с данными. В качестве примера рассмотрим исходный код Koin-модуля с данными:

```
val dataModule = module {
    single<DataRepository> {
        DataRepositoryImpl(
            buildingItemsDatabase = get(), service = get(),
            buildingItemJsonMapper = get(), buildingItemDBMapper = get(),
            structuralObjectItemJsonMapper = get(), structuralObjectItemDBMapper = get(),
            addressItemJsonMapper = get(), addressItemDBMapper = get(),
            iconItemJsonMapper = get(), iconItemDBMapper = get()
        )
    }
    single {
        fun buildDatabase(context: Context) =
            Room.databaseBuilder(
                context.applicationContext,
                BuildingItemsDatabase::class.java, "bsumapDB"
```

```
            ) // Use code below and your Migration if you've made some changes in
DB structure
                .addMigrations(MigrationDB.MIGRATION_1_2)
                //.addMigrations(MigrationDB.MIGRATION_2_3)
                .build()
        buildDatabase(androidContext())
    }
    single {
        val retrofitClient: RetrofitClient = get()

        retrofitClient.retrofit("http://map.bsu.by:51107/")
            .create(MapDataApi::class.java)
    }
    single { RetrofitClient(client = get()) }
    single {
        OkHttpClient()
            .newBuilder()
            .addInterceptor(HttpLoggingInterceptor().apply {
                level = HttpLoggingInterceptor.Level.BODY
            })
            .build()
    }
```

```
    single { BuildingItemJsonMapper() }

    single { BuildingItemDBMapper() }

    single { StructuralObjectItemJsonMapper() }

    single { StructuralObjectItemDBMapper() }

    single { AddressItemJsonMapper() }

    single { AddressItemDBMapper() }

    single { IconItemJsonMapper() }

    single { IconItemDBMapper() }

}
```

Как можно заметить, в самом начале данного модуля создается объект репозитория – места, где хранятся все поля и методы работы с данными приложения. Для инициализации того или иного объекта требуется указывать *get()* метод для всех его параметров (если таковые имеются). Ключевое слово single используется, чтобы указать, что объект необходим лишь в одном экземпляре.

Далее следует создание объекта базы данных. Как можно заметить, внутри объекта вызывается уже функция. В эту функцию передаются необходимые параметры базы данных (контекст, название, миграции). Другими словами, при необходимости, для некоторых объектов вызываются функции прямо в Koin-модуле.

Далее создается объект Retrofit для того, чтобы получать данные с сервера. Внутри его области видимости создана переменная. Эта переменная предварительно создано прямо в Koin-модуле, чтобы корректно инициализировать объект. Таким образом, можно даже создавать переменные для инициализации объектов.

Все остальные объекты создаются по аналогичному принципу.

На картинке ниже можно видеть, как в классе *WorkManagerApplication* модуля *app* вызываются вышеупомянутые Koin-модули (см. рис. 2.3):

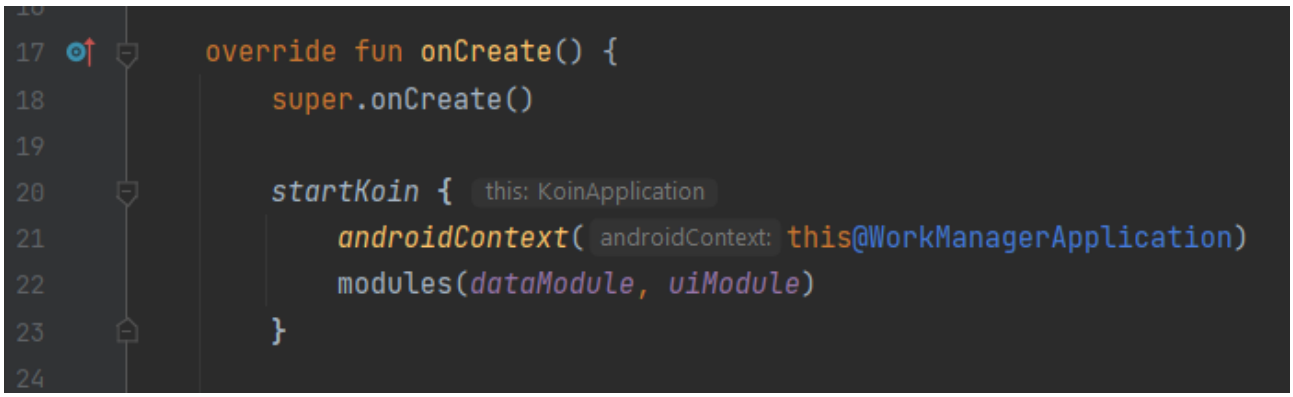

Рисунок 2.3 – Запуск предварительно созданных Koin-модулей

Таким образом, в проекте успешно было реализовано внедрение зависимостей, что уменьшило вероятность ошибок и ускорило компиляцию исходного кода.

2.5 Получение данных с сервера

В качестве сервера с данными для приложения выступает домент: http://map.bsu.by/. Данный домен активно разрабатывается разработчиком проекта, отвечающим за веб-приложение и сервер. Задачей же этого проекта являлось получение данных с этого домена и последующее их отображение на интерфейсе мобильного приложения пользователя.

Для того, чтобы успешно достать данные с сервера, было принято решение использовать технологии Retrofit и Okhttp.

Для выполнения поставленной задаче в модуле с данными *data* в папке *retrofit* был создан класс *RetrofitClient* и интерфейс *MapDataApi*.

Ниже представлен исходный код класса *RetrofitClient*:

```
class RetrofitClient(private val client: OkHttpClient) {

    fun retrofit(baseUrl : String) : Retrofit = Retrofit.Builder()

        .client(client)

        .baseUrl(baseUrl)

        .addConverterFactory(GsonConverterFactory.create())

        .build()
```

}

Данный класс имеет функцию *retrofit()*, которая принимает на вход несколько параметров.

Во-первых, на вход подается клиент Okhttp, который передаётся в качестве параметра класса. Клиент Okhttp по сути представляет собой оболочку интерфейса *MapDataApi*, о котором будет сказано далее.

Помимо этого на вход также подаётся базовый URL домена (в данном случае, это будет следующий URL: http://map.bsu.by:51107/). Стоит отметить, что при переходе на этот URL данные получить не получится, так как это лишь часть необходимой ссылки. Данный URL является лишь составной компонентой, а остальные компоненты выбираются в зависимости от того, какие конкретно данные требуется запросить от сервера.

На вход функции также передаётся необходимый тип конвертера данных. В случае проекта, форматом выступил GSON, соответственно для него был подобран конвертер *GsonConverterFactory*.

Далее представлен исходный код интерфейса *MapDataApi*:

```
interface MapDataApi{

    @GET("api/buildings/all")

    suspend fun getData() : List<BuildingItemJson>?

    @GET("api/building-photos")

    suspend fun getImagesWithBuildingId(@Query("buildingId") buildingId: String) : List<BuildingItemImageJson>

}
```

Данный интерфейс содержит в себе @GET() методы, внутри которых содержатся компоненты ссылки на нужные данные. Например, в вышеуказанном исходном коде первый метод получает данные абсолютно обо всех объектах и возвращает их в виде списка сущностей JSON формата, а второй метод получается данные об изображениях объектов и возвращает их в виде соответствующего списка сущностей JSON формата. При конкатенации данных компонент ссылок с базовым URL получается ссылка, при переходе на которую приложение имеет возможность загружать данные с сервера.

В качестве сущностей JSON данных был разработан ряд классов. Одним из них является *BuildingItemJson*, исходный код которого представлен ниже:

```
data class BuildingItemJson(
```

@SerializedName("structuralObjects") val structuralObjects: List<StructuralObjectItemJson?>?,

@SerializedName("id") var id: String?,

@SerializedName("inventoryUsrreNumber") var inventoryUsrreNumber: String?,

@SerializedName("name") var name: String?,

@SerializedName("isModern") var isModern: String?,

@SerializedName("address") var address: BuildingItemAddressJson?,

@SerializedName("type") var type: BuildingItemTypeJson?

)

Как можно заметить, данный класс содержит в себе поля, которые принимают данные по определенному ключу и сохраняют их либо в виде примитивных типов (String, Integer, Boolean), либо в виде таких же составных классов. Далее эти данные конвертируются в сущности базы данных и сохраняются локально.

Таким образом, с помощью технологий Retrofit и Okhttp была реализована загрузка данных для приложения с веб-сервера.

2.6 Создание и наполнение локальной базы данных

2.6.1 Создание сущностей базы данных

Далее среди целей проекта было создание локальной базы данных, чтобы обеспечить кэширование и, соответственно, более высокий уровень запуска и производительности.

Первым делом, требовалось создать сущности базы данных. Для этого было создано 5 классов, представляющих из себя сущности объектов для базы данных: *BuildingItemEntityDB*, *StructuralObjectItemEntityDB*, *AddressItemEntityDB, BuildingItemImageEntityDB, IconEntityDB*. Данным классам соответствуют таблицы в базе данных. В случае, если требуется получить объект с данными, содержащим в себе поля из разных таблиц, создаётся отдельный класс, о чём будет упомянуто далее.

В качестве примера, рассмотрим исходный код *BuildingItemEntityDB*:

```
@Entity(tableName = "buildings")
data class BuildingItemEntityDB(
    @PrimaryKey
    val id: String,
    @ColumnInfo(name = "inventoryUsrreNumber")
    val inventoryUsrreNumber: String?,
    @ColumnInfo(name = "name")
    val name: String?,
    @ColumnInfo(name = "isModern")
    val isModern: Boolean?,
    @ColumnInfo(name = "type")
    val type: String?,
    @ColumnInfo(name = "markerPath")
    val markerPath: String?
)
```

Внутри ключевого слова *@Entity*, означающего, что мы работаем именно с сущностью, вписывается название таблицы. Например, в вышеуказанном случае, имя таблицы – «buildings». Далее перечисляются поля таблицы.

Как правило, все поля таблицы представляют из себя примитивные типы. В случаях, когда требуется какой-то нестандартный тип (например, дата), используются специальные конвертеры с ключевым словом @TypeConverter (см. рис. 2.4):

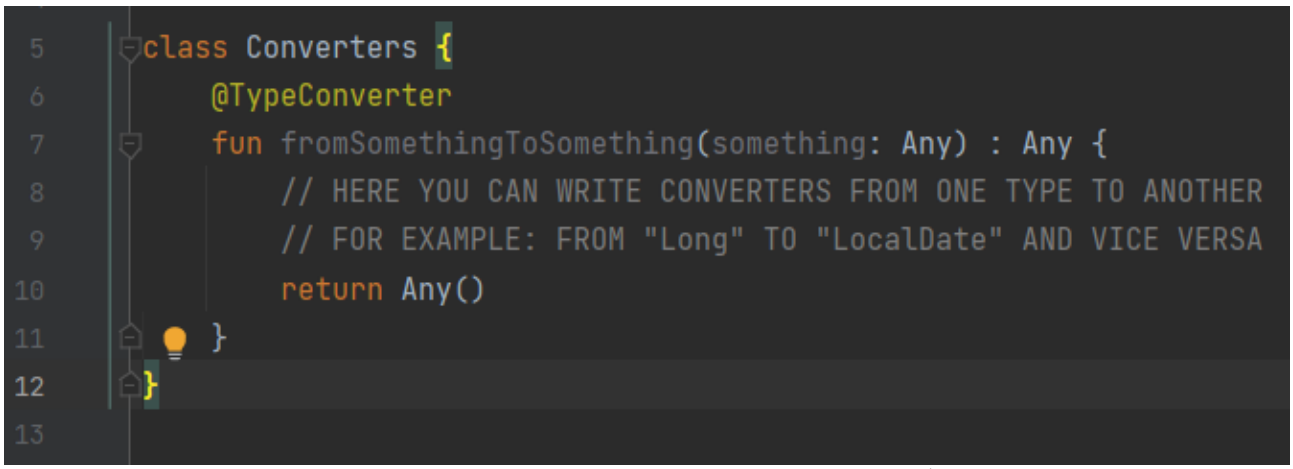

Рисунок 2.4 – Пример конвертера для конвертации типов в базе данных

Остальные классы сущностей построены аналогичным образом.

Однако хорошей практикой является разделять логику базы данных от доменной логики, тем самым скрывая детали реализации на одном уровне абстракции от другого. По этой причине в проекте было принято решение создать на каждый объект не только сущность базы данных в модуле *data*, но и доменные сущности в модуле *domain*. Таким образом, были созданы следующие доменные сущности: *BuildingItem*, *StructuralObjectItem*, *AddressItem*, *BuildingItemImage*, *IconImage*. В качестве примера приведём исходный код класса *BuildingItem*:

data class BuildingItem(

val id: String,

val structuralObjects: List<StructuralObjectItem?>?,

val imagesList: List<BuildingItemImage?>?,

val inventoryUsrreNumber: String?,

val name: String?,

val isModern: Boolean?,

val address: AddressItem?,

val type: String?,

val markerPath: String?

)

Как можно заметить, класс доменной сущности отличается от класса сущности базы данных. Дело в том, что в доменном объекте мы иногда желаем видеть другие объекты (или даже списки с другими объектами). На уровне базы данных это сделать невозможно, так как каждая таблица может иметь среди полей лишь примитивные типы или конвертеры. По этой причине для некоторых сущностей базы данных создаются дополнительные классы. В частности, в этом проекте доменная сущность *BuildingItem* содержит в себе список сущностей *StructuralObjectItem* и список сущностей *BuildingItemImage*, а также объект сущности *AddressItem*, соответственно для получения этой сущности не будет достаточно преобразовать лишь сущность базы данных *BuildingItemEntityDB*, ибо нам нужна целая комбинация сущностей. Поэтому в модуле *data* были также добавлены следующие классы: *BuildingItemDB* и *StructuralObjectItemDB*.

В качестве примера рассмотрим исходный код класса *BuildingItemDB*:

```
data class BuildingItemDB (
    @Embedded
    val buildingItemEntityDB: BuildingItemEntityDB,
    @Relation(
        parentColumn = "id",
        entityColumn = "buildingItemId"
    )
    val structuralObjectEntities: List<StructuralObjectItemEntityDB>?,
    @Relation(
        parentColumn = "id",
        entityColumn = "buildingItemId"
    )
    val buildingItemImageEntities: List<BuildingItemImageEntityDB>?,
```

```
    @Relation(
        parentColumn = "id",
        entityColumn = "buildingItemId"
    )
    val iconEntities: List<IconItemEntityDB>?,
    @Relation(
        parentColumn = "id",
        entityColumn = "buildingItemId"
    )
    val address: AddressItemEntityDB?
)
```

Как можно заметить, данный класс включает в себя ранее упомянутую сущность базы данных *BuildingItemEntityDB*, а также несколько списков с объектами других сущностей, обозначенных ключевыми словами *@Relation*. Другими словами, мы связываем несколько таблиц (по их идентификаторам) для того, чтобы получить нужную комбинацию данных, которая позже конвертируется в доменную сущность. В нашем случае мы можем взять не только поля из таблицы *buildings*, но и некоторые поля из таблиц *structuralObjects*, *addresses*, *buildingItemImages* и *icons*, чтобы получить желаемую комбинацию данных. Затем, именно объект этого класса преобразуется в доменную сущность *BuildingItem*. С другими сущностями работа проходит аналогичным образом.

2.6.2 Создание DAO интерфейсов

Для того, чтобы работать с методами таблиц базы данных, создаются специальные DAO интерфейсы. Они позволяют собрать в одном месте необходимые CRUD методы для сущности (таблицы базы данных). В качестве примера рассмотрим интерфейс *IconItemDAO*:

```
@Dao
```

```
interface IconItemDAO {

    @Insert(onConflict = OnConflictStrategy.REPLACE)

    suspend fun insertOneIconItem(item: IconItemEntityDB): Long

    @Insert(onConflict = OnConflictStrategy.REPLACE)

    suspend fun insertIconItemsList(items: List<IconItemEntityDB>):
List<Long>

    @Update

    suspend fun updateOneIconItem(item: IconItemEntityDB)

    @Update

    suspend fun updateAllIconItems(items: List<IconItemEntityDB>)

    @Delete

    suspend fun deleteOneIconItem(item: IconItemEntityDB)

    @Delete

    suspend fun deleteAllIconItems(items: List<IconItemEntityDB>)

    @Query("SELECT * FROM icons")

    fun getAllIconItems(): Flow<List<IconItemEntityDB>>

    @Query("SELECT * FROM icons WHERE id = :neededId")

    fun getIconItemById(neededId: String): Flow<IconItemEntityDB>

}
```

В методах интерфейсов DAO могут использоваться разные ключевые слова: *@Insert, @Update, @Delete, @Query*.
Ключевое слово *@Insert* используется в тех случаях, когда необходимо обеспечить вставку в таблицу базы данных. Для этого на вход подаётся объект сущности базы данных, а на выходе функция возвращает число, уведомляющее

о результате операции. Для случаев, когда данные уже существуют в таблице, существует несколько стратегий. Одной из них является замещение старых данных на новые. Именно эта стратегия используется в данном проекте. Данная стратегий используется базой данной при написании следующей инструкции: *@Insert(onConflict = OnConflictStrategy.REPLACE)*

Ключевое слово *@Update* используется для того, чтобы обновить существующие данные в таблице базы данных. Как и в случае со вставкой, для этого на вход подаётся объект сущности базы данных, а на выходе функция возвращает число, уведомляющее о результате операции.

Ключевое слово *@Delete* используется для того, чтобы удалить существующие данные в таблице базы данных. Как и в двух вышеупомянутых случаях, для этого на вход подаётся объект сущности базы данных, а на выходе функция возвращает число, уведомляющее о результате операции.

Ключевое слово *@Query* используется для создания запросов с помощью синтаксиса языка SQL. В частности, в данном проекте это ключевое слово использовалось для выборки данных. Запрос *SELECT * FROM icons* позволяет вывести список всех имеющихся в базе данных объектов иконок. Запрос *SELECT * FROM icons WHERE id = :neededId* позволяет вывести объект иконки, идентификатор которой соответствует заданному.

Аналогичные методы реализованы и для остальных сущностей базы данных.

2.6.3 Создание конвертеров

В этом подразделе хотелось бы упомянуть несколько деталей, касающихся конвертеров.

В начале требовалось создать конвертеры, которые бы преобразовывали сущности JSON формата в сущности базы данных, а затем – конвертеры, которые бы преобразовывали сущности базы данных в доменные сущности.

Для решения первой задачи (преобразования из JSON формата в формат сущностей базы данных) были написаны следующие классы: *BuildingItemJsonMapper*, *StructuralObjectItemJsonMapper*, *AddressItemJsonMapper*, *BuildingItemImageJsonMapper*, *IconItemJsonMapper*. Данные классы позволяют конвертировать сущность JSON формата (которая образуется при получении данных от веб-сервера) в сущность базы данных. В качестве примера приведём исходный код *IconItemJsonMapper*:

```
// This mapper converts a JSON entity to a database entity

class IconItemJsonMapper {
```

```
    fun fromJsonToRoomDB(itemJson: StructuralObjectItemIconJson?,
structuralObjectItemId: String?,
                         buildingItemId: String?) : IconItemEntityDB?
    {
        return if (itemJson == null) {
            null
        } else {
            IconItemEntityDB(itemJson.structuralObjectId!!,
                structuralObjectItemId,
                itemJson.subdivision,
                itemJson.logoPath,
                buildingItemId)
        }
    }
}
```

Здесь видно, что класс имеет один единственный метод, который принимает на вход необходимый объект формата JSON, а также необходимые идентификаторы (от других сущностей), после чего возвращает объект сущности базы данных. В случае, если объект формата JSON пришёл пустым, возвращается значение *null*.

После того, как сущности базы данных и их DAO интерфейсы были готовы, возникла необходимость в их конвертации в доменные сущности для дальнейшей работы. Для этого были созданы следующие классы: *BuildingItemDBMapper*, *StructuralObjectItemDBMapper*, *AddressItemDBMapper*, *BuildingItemImageDBMapper*, *IconItemDBMapper*. Данные классы нужны за тем, чтобы преобразовать имеющиеся сущности базы данных в доменные сущности. В качестве примера приведём исходный код класса *IconItemDBMapper*:

```
// This mapper converts a database entity to a domain entity
class IconItemDBMapper {
    fun fromDBToDomain(item: IconItemEntityDB?) : IconItem? {
        return if (item == null) {
            null
        } else {
            IconItem(item.id,
                item.subdivision,
                item.logoPath)
        }
    }
}
```

Здесь видно, что класс имеет один единственный метод, который принимает на вход необходимый объект базы данных (который в свою очередь может являться комбинацией из нескольких сущностей), после чего возвращает объект доменной сущности. В случае, если объект базы данных пришёл пустым, возвращается значение *null*.

2.6.4 Миграции базы данных

После того, как база данных была успешно создана (см. рис. 2.5), требовалось подумать об интерфейсе, который позволил бы быстро и эффективно изменить структуру базы данных. По этой причине были изучены возможности внедрения миграций для базы данных.

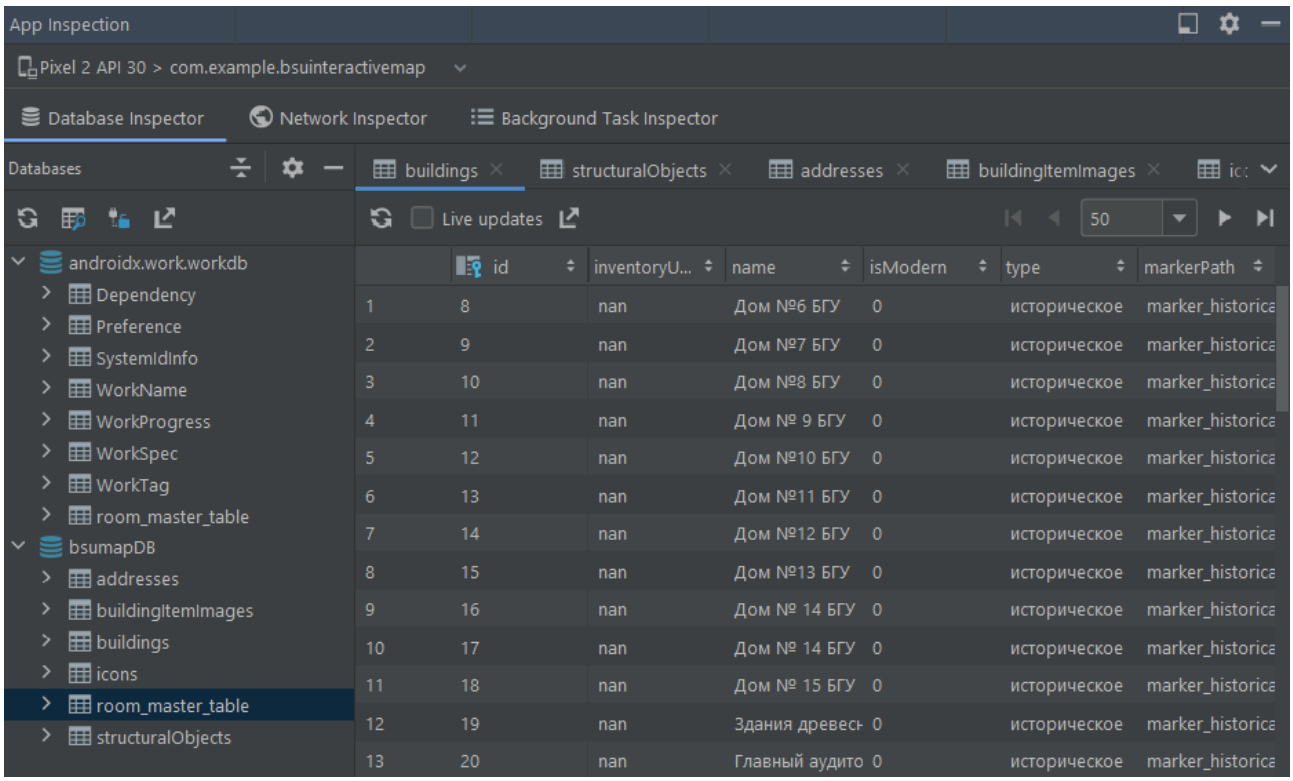

Рисунок 2.5 – Успешное создание и заполнение базы данных

Для того, чтобы внести какие-либо изменения в структуру базы данных (например, добавить таблицу или изменить какое-то поле существующей таблицы), требуется написать SQL-запрос, поместив его в объект миграции. Ниже приведён исходный код объекта *MigrationDB*:

```
object MigrationDB{

    val MIGRATION_2_3 = object : Migration(2, 3) {

        override fun migrate(db: SupportSQLiteDatabase) {

            // HERE YOU CAN CHANGE DATABASE STRUCTURE (don't forget to add it at KoinModule)

            db.execSQL("")

        }

    }

    val MIGRATION_1_2 = object : Migration(1, 2) {

        override fun migrate(db: SupportSQLiteDatabase) {
```

```
                db.execSQL("CREATE  TABLE  IF  NOT EXISTS buildingItemImages ("
+
                        "id TEXT PRIMARY KEY NOT NULL, " +
                        "description TEXT, " +
                        "imagePath TEXT, " +
                        "buildingItemId TEXT);")
            }
        }
    }
```

Так получилось, что в проекте пришлось однажды использовать одну миграцию (а именно требовалось добавить новую таблицу для хранения сущностей с путями картинок). Поэтому рассмотрим миграцию на этом примере.

У каждой миграции должно быть два входных параметра, первый из которых символизирует версию базы данных до использования этой миграции, а второй – символизирует версию базы данных после использований этой миграции. Например, в нашем случае в качестве первого параметра подаётся 1, а в качестве второго параметра – 2. Это означает, что, при выполнении SQL-запроса, связанного с этой миграцией, база данных сменит свою версию с первой на вторую. Иногда удобно заготовить такие миграции, которые позволяют повысить версию базы данных сразу до нужной (например, с первой до третьей), однако для этого понадобится написать дополнительные объекты миграций.

Ниже приведён фрагмент кода, в котором созданная миграция добавляется к текущей базе данных:

```
    fun buildDatabase(context: Context) =
            Room.databaseBuilder(
                context.applicationContext,
                BuildingItemsDatabase::class.java, "bsumapDB"
            ) // Use code below and your Migration if you've made some changes in
DB structure
```

```
.addMigrations(MigrationDB.MIGRATION_1_2)

//.addMigrations(MigrationDB.MIGRATION_2_3)

.build()
```

Таким образом, удалось реализовать локальную базу данных в проекте путём создания различных сущностей базы данных, соответствующих им DAO интерфейсов, а также конвертеров из одних сущностей в другие и объектов миграции для будущих потенциальных изменений в структуре.

2.7 Создание фрагмента интерактивной карты

Следующей целью проекта было создание фрагмента для расположения интерактивный карты в нём. Для этого в модуле пользовательского интерфейса *ui* был создан ряд классов: *MainActivity, MapFragment*, *HistBuildingDetailsFragment*, *ModernDepartDetailsFragment*, *DepartmentsListAdapter*, *ImagesPagerAdapter*, *MapViewModel*, *LoadStatem LocationPermissionHelper*, *KoinUiModule*.

Листинг класса MapFragment, в котором происходят ключевые действия по созданию интерактивной карты, представлен в приложении (см. приложение А).

Ниже приведён фрагмент кода класса *MainActivity*, описывающий метод *onCreate()*:

```
override fun onCreate(savedInstanceState: Bundle?) {

    super.onCreate(savedInstanceState)

    binding = ActivityMainBinding.inflate(layoutInflater)

    val view = binding.root

    setContentView(view)

    if (savedInstanceState == null)

    {

       supportFragmentManager.beginTransaction()
```

```
            .replace(R.id.fragment_container, MapFragment())
            .commit()
        }
    }
```

Класс *ActivityMainBinding* является сгенерированным на основе ресурса *activity_main.xml*. Технология Binding позволяет работать с view-представлениями с помощью специально сгенерированных классов, что гораздо удобнее и надёжнее, чем пользоваться методом *findViewById()*, так как исключается вероятность возникновения исключения вида *NullPointerException()*. Данная технология используется в том числе и для представлений фрагментов проекта.

Для наглядности ниже приведена разметка *activity_main.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
```

```
app:layout_constraintEnd_toEndOf="parent"

app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toTopOf="parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Во время вызова метода *replace(R.id.fragment_container, MapFragment())* фрагмент карты раздувается в ячейку с адресом *R.id.fragment_container*, после чего происходит создание фрагмента карты.

2.8 Размещение объектов на интерактивной карте

Когда фрагмент карты был создан, было необходимо разместить объекты на карте. Для этого требовалось создать несколько сущностей технологии Mapbox, а именно *PointAnnotationManager* и *ViewAnnotationManager*.

Помимо объектов на карте также было необходимо разместить текущее положение пользователя, однако для этого требовалось запросить специальное разрешение. С этой целью был создан класс *LocationPermissionHelper*. Ниже приведён исходной код его функции *checkPermission()*, которая запрашивает у пользователя разрешение на отслеживание геолокации:

```
private lateinit var permissionsManager: PermissionsManager

fun checkPermissions(onMapReady: () -> Unit) {

    if (PermissionsManager.areLocationPermissionsGranted(activity.get())) {

        onMapReady()

    } else {

        permissionsManager = PermissionsManager(object :
PermissionsListener {

            override fun onExplanationNeeded(permissionsToExplain:
List<String>) {

                Toast.makeText(
```

```
                activity.get(), "You need to accept location permissions.",
                Toast.LENGTH_SHORT
            ).show()
        }

        override fun onPermissionResult(granted: Boolean) {
            if (granted) {
                onMapReady()
            } else {
                activity.get()?.finish()
            }
        }
    })
    permissionsManager.requestLocationPermissions(activity.get())
  }
}
```

Запрос разрешения у пользователя происходит лишь в том случае, если пользователь ещё ни разу не давал доступ или если он давал доступ временно. Если пользователь отказывает в доступе, Activity завершает свою работу. Если же пользователь даёт доступ, то фрагмент карты продолжает выполнять дальнейшие инструкции. Запрос разрешения на отслеживание геолокации у пользователя представлен на рисунке ниже (см. рис. 2.6):

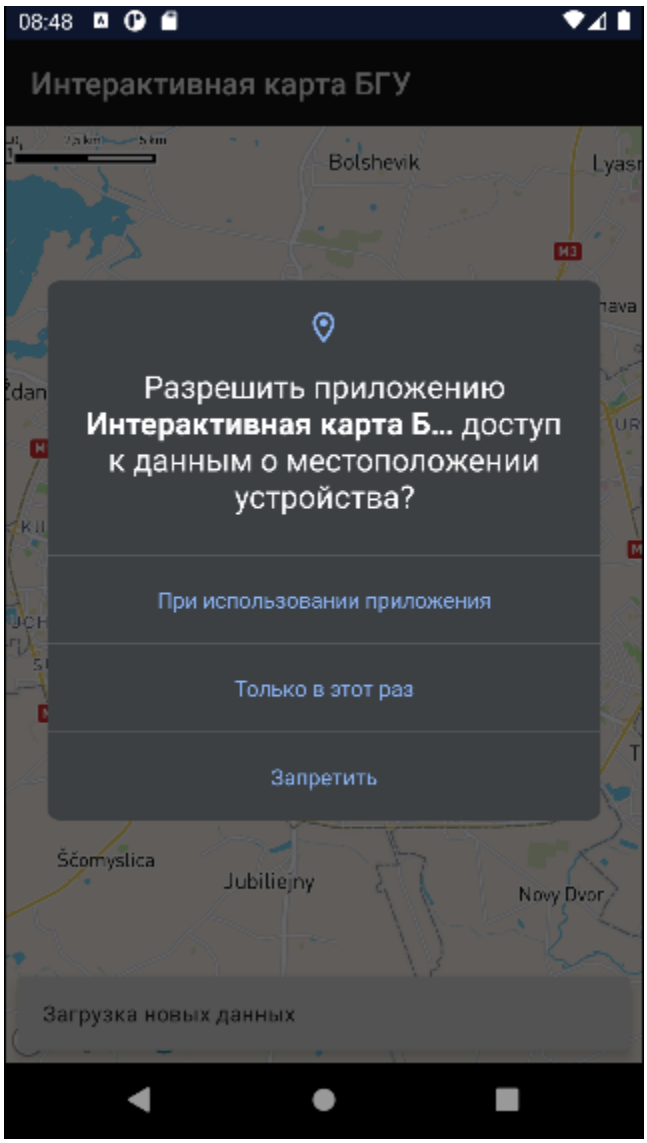

Рисунок 2.6 – Запрос у пользователя разрешения на отслеживание геолокации

Далее фрагменту необходимо загрузить данные из класса *MapViewModel*. Класс *MapViewModel* отвечает за загрузку данных из репозитория, который в свою очередь находится в модуле *data* и достает данные из локальной базы данных, а также с веб-сервера, если имеются новые данные. Ниже приведена одна из функций класса *MapViewModel*, отвечающая за загрузку новостей из репозитория, – *loadData()*:

```
// This function tries to update old data to actual data if possible

fun loadData() {

viewModelScope.launch(Dispatchers.Main) {

state.value = LoadState.LOADING
```

```
            try {
                dataRepository.loadData()
                state.value = LoadState.SUCCESS
            } catch (e: Throwable) {
                if (!isConnectedToInternet() && e is IOException) {
                    state.value = LoadState.INTERNET_ERROR
                }
                else if (e is NullPointerException) {
                    state.value = LoadState.EMPTY_DATA_ERROR
                }
                else {
                    state.value = LoadState.UNKNOWN_ERROR
                }
            }
        }
    }
```

В случае, если данные из репозитория по каким-то причинам не были загружены, состояние *LoadState* примет соответствующий код ошибки, который потом дойдёт до пользователя в виде уведомления. Это могут быть: ошибки Интернет-соединения, ошибки пустого объекта, неизвестные ошибки. Ниже приведён исходный код функции *loadData()*, которая находится в классе репозитория:

```
    // This function is needed to get actual data from server
    override suspend fun loadData() {
```

```
            try {
                val items = service.getData()
                    ?.filter { isItemWithID(it) } // Clean list from items with null id
                    ?.map {
                        val itemsImages : List<BuildingItemImageJson> =
                            service.getImagesWithBuildingId(it.id!!)
                        BuildingItemJsonMapper().fromJsonToRoomDB(it,
itemsImages)!!
                    }
                    ?.filter { isItemNotEmpty(it) } // Clean DB from items with empty data

buildingItemsDatabase.buildingItemsDao().insertBuildingItemsList(items!!)
            } catch (e: Throwable) {
                throw NullPointerException("Error: " +
                        "Some BuildingItem (or even whole list) from json is empty!\n" +
e.message)
            }
        }
```

Однако прежде, чем загружать новые данные с сервера, *MapViewModel* сначала вызывает метод *getItems()* класса репозитория, тем самым располагая на карте объекты, которые уже есть в локальной базе данных (тем самым ускоряя расстановку объектов). Листинг метода *getItems()* представлен ниже:

```
    // This function return list of all items that are in local database
        override fun getItems(): Flow<List<BuildingItem>> {
```

```
        try {

            return
buildingItemsDatabase.buildingItemsDao().getAllBuildingItems().map { list ->

                list.map { BuildingItemDBMapper().fromDBToDomain(it)!! } }

        } catch (e: Throwable) {

            throw NullPointerException("Error while getting building items: " +

            "Some item is nullable!\n" + e.message)

        }

    }
```

Стоит отметить, что обе функции являются *suspend-функциями*. Другими словами, эти функции выполняются не в основном потоке приложения, а в дополнительных, которые создаются динамически. Соответственно для получения данных используется контейнер *Flow*, который позволяет получать потоки данных в многопоточных приложениях. Как только поступают новые данные, метод *collect{...}* контейнера *Flow* способен вернуть их, тем самым обеспечения асинхронное получение данных.

Как только данные получены (пусть даже из локальной базы данных, не обязательно от веб-сервера), вызывается метод *setMarkers()*, который отвечает за расстановку объектов на карте. Этот метод также имеется в листинге класса *MapFragment* (см. приложение А). В этот метод подаётся список объектов, представляющих из себя сущности зданий. Прежде, чем добавить точку объекта на карту, необходимо подготовить её аннотацию. Для этого требуется задать несколько параметров: широту точки, долготу точки, ресурс для иконки. После этого можно вызывать метод *create(pointAnnotationOptions)* класса *PointAnnotationManager*, после чего точка будет успешно поставлена на карте.

На картинке ниже представлен момент расстановки точек объектов на интерактивной карте приложения (см. рис. 2.7):

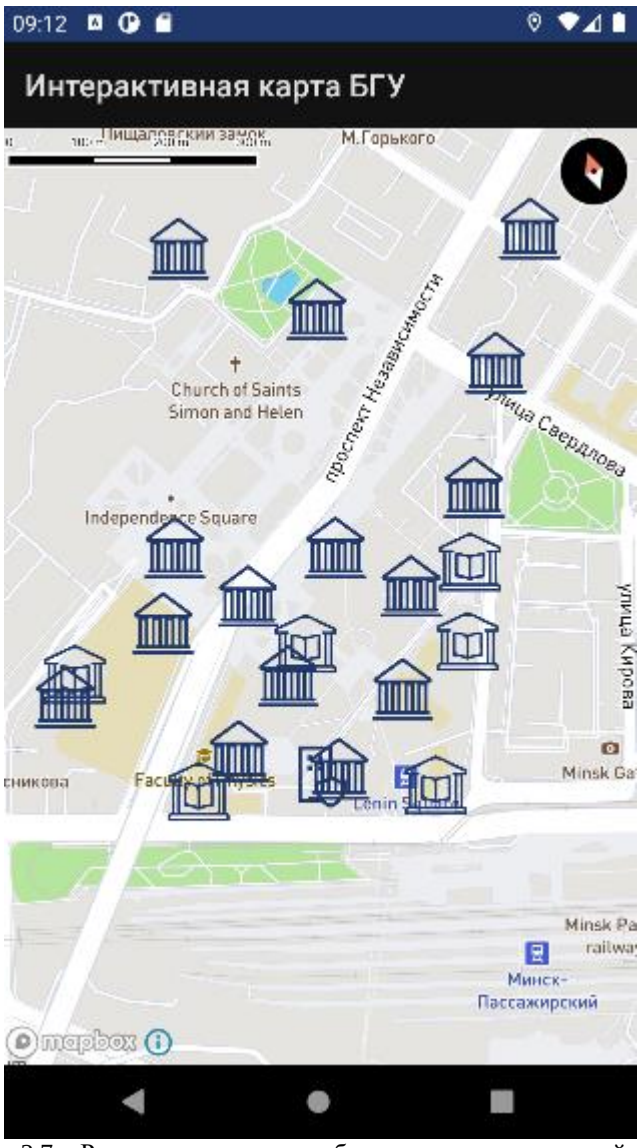

Рисунок 2.7 – Расположение точек объектов на интерактивной карте

Таким образом, удалось успешно расставить точки всех объектов на интерактивной карте. Далее требовалось создать аннотации для этих точек.

2.9 Создание и привязка аннотаций для фрагментов

Согласно специфике проекта, все достопримечательности делились на два вида: исторические и современные. При этом в современных зданиях могло располагаться более одного департамента (например, в главном корпусе БГУ располагается не только механико-математический факультет, но и факультет прикладной математики и информатики). Поэтому было принято решение создать две разные разметки и два разных класса фрагмента для исторических и современных зданий.

Представление аннотации для исторических зданий представляет из себя диалоговое окно с одним единственным департаментом (см. рис. 2.8):

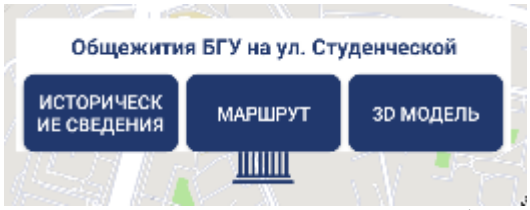

Рисунок 2.8 – Аннотация исторического объекта

Представление аннотации для современных зданий, вообще говоря, представляет из себя диалоговое окно с несколькими департаментами (см. рис. 2.9):

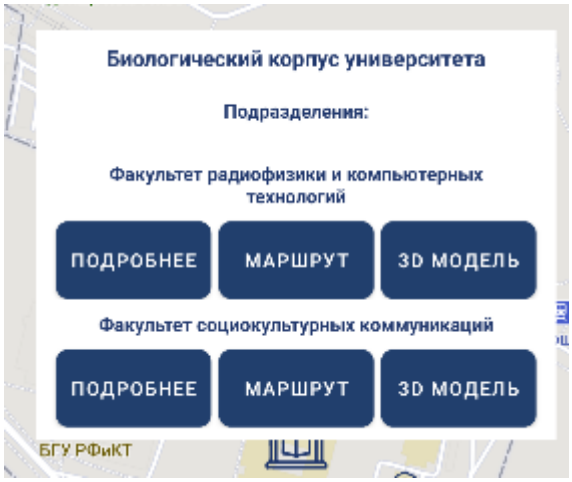

Рисунок 2.9 – Аннотация современного объекта

Вышеуказанные фрагменты требовалось разместить в качестве аннотаций. На картинке выше можно заметить кнопки «Маршрут» и «3D Модель». Стоит отметить, что на данной стадии разработки в проекте ещё не реализованы эти две функции. Данные кнопки заготовлены на будущее, для случая модификации проекта и добавления в него новых возможностей. Если попробовать нажать на одну из этих кнопок в данный момент, то высветится сообщение о том, что данные функции будут разработаны в будущем.

Если объект – исторический, то происходит прикрепление аннотации к текущей точке, а также раздувание фрагмента с помощью инструкции *HistBuildInfoBinding.bind(viewAnnotation).apply{…}*. Внутри блока *aply{…}* происходит обработка элементов аннотации, включая кнопки. В этом блоке создаётся слушатель для кнопки, по которой пользователь может перейти к фрагменту с деталями и галереей изображений, связанных с объектом. Об этом фрагменте речь пойдёт в следующем разделе.

Если же объект – современный, то придётся сначала вызвать специальный класс *DepartmentsListAdapter*, который необходим, чтобы создать список, состоящий из отдельных аннотаций для каждого департамента. Ниже

представлен исходный код метода *onBindViewHolder()*, располагающегося внутри класса адаптера:

```
override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    val departmentObject = getItem(position)
    holder.departmentPreviewTitle.text = departmentObject.subdivision
    holder.dpBtnSeeDetails.setOnClickListener {
        activity.onModernBuildingClick(departmentObject, imageList)
    }
    holder.dpBtnCreateRoute.setOnClickListener {
        createSnackbar(context.resources.getString(R.string.future_feature),
            context.getColor(R.color.black))
    }
    holder.dpBtnSee3DModel.setOnClickListener {
        createSnackbar(context.resources.getString(R.string.future_feature),
            context.getColor(R.color.black))
    }
}
```

Данный метод необходим, чтобы обработать аннотацию для каждого департамента из списка, принадлежащего современному объекту. Внутри этого метода, как и в случае с историческими объектами, также создается слушатель для кнопки. Соответственно, при нажатии на кнопку пользователь сможет открыть отдельный фрагмент с информацией о департаменте, а также с галереей изображений этого департамента.

2.10 Создание фрагментов с информацией об объектах

Среди методов *MainActivity* заготовлены следующие два метода:

```
fun onHistoricalBuildingClick(building: BuildingItem, imagesList:
List<BuildingItemImage?>?) {

    inflateFragment(HistBuildingDetailsFragment.newInstance(building,
imagesList),

        R.id.fragment_container,true)

}

fun onModernBuildingClick(department: StructuralObjectItem, imagesList:
List<BuildingItemImage?>?) {


inflateFragment(ModernDepartDetailsFragment.newInstance(department,
imagesList),

        R.id.fragment_container,true)

}
```

Эти методы заменяют текущий фрагмент карты на фрагмент с информацией того или иного объекта в случаях, когда пользователь нажал соответствующую кнопку. Структура данных фрагментов очень схожа, потому разберём её на примере исторических объектов. Ниже приведён исходный код метода *onViewCreated()* класса фрагмента *HistBuildingDetailsFragment*:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

    super.onViewCreated(view, savedInstanceState)

    val building = arguments?.getParcelable<BuildingItem>(buildingID)

    val imagesList =
arguments?.getParcelableArrayList<BuildingItemImage>(ModernDepartDetailsFra
gment.imagesListId)?.toList()

    val pageTitle: TextView = binding.title

    val pageImgPager: ViewPager2 = binding.imgPager
```

```
        val pageText: TextView = binding.info

        pageTitle.text = Html.fromHtml(building?.name,
Html.FROM_HTML_MODE_LEGACY).toString()

        val adapter = ImagesPagerAdapter(false)

        adapter.submitList(imagesList)

        pageImgPager.adapter = adapter

        pageText.text = Html.fromHtml(building?.address?.description,
Html.FROM_HTML_MODE_LEGACY).toString()

    }
```

Внутри этого момента происходит присваивание значений с информацией об объекте соответствующим ресурсам. Однако один из компонентов требует более подробного рассмотрения.

Для каждого объекта, вообще говоря, прилагается целый список изображений, которые с ним ассоциируются. Поэтому было принято решение реализовать демонстрацию данных изображений с помощью технологии *ViewPager2*. Данная технология позволяет пользователю прокручивать изображения слева-направо и справа-налево, что достаточно удобно. Однако для данной технологии потребовалось реализовать дополнительный класс – *ImagesPagerAdapter*. Исходный код данного класса приведён ниже:

```
    // This class is needed to place image objects (consist of image and
description) into ViewPager

    class ImagesPagerAdapter(private val isModern: Boolean) :

        ListAdapter<BuildingItemImage,
ImagesPagerAdapter.ItemViewHolder>(ImageObjectsDiffCallback()) {

        class ItemViewHolder(itemBinding: ImgPagerItemBinding) :

        RecyclerView.ViewHolder(itemBinding.root) {

        val image: ImageView

        val imgDescription: TextView
```

```
            init {
                image = itemBinding.pagerImg
                imgDescription = itemBinding.info
            }
        }
        override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
            val imageObject = getItem(position)
            if (isModern) {
Picasso.get().load("http://map.bsu.by/buildings_images/modern_buildings/" +
                        imageObject.imagePath).into(holder.image)
            } else {
Picasso.get().load("http://map.bsu.by/buildings_images/historical_buildings/" +
                        imageObject.imagePath).into(holder.image)
            }
            holder.imgDescription.text = Html.fromHtml(imageObject.description,
Html.FROM_HTML_MODE_LEGACY).toString()
        }
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ItemViewHolder {
            return
ItemViewHolder(ImgPagerItemBinding.inflate(LayoutInflater.from(parent.context),
parent, false))
```

```
    }

}
```

Данный класс по своей структуре очень сход с адаптерами, которые активно используются для представлений списков – *RecycleView*.

Так как картинки не могут храниться в проекте локально из-за ограничения по памяти, было принято решения загружать их с сервера по ссылке. Для этой цели была применена технология *Picasso*. Как можно заметить выше, с помощью метода *Picasso.get().load(...).into(holder.image)* приложение загружает картинки на устройство пользователя динамически, что положительно сказывается на аспекте экономии памяти мобильного устройства, потому что не приходится хранить картинки локально.

Ниже представлен фрагмент одного из современных департаментов с галереей изображений после момента, когда пользователь нажал на соответствующую кнопку (см. рис. 2.10):

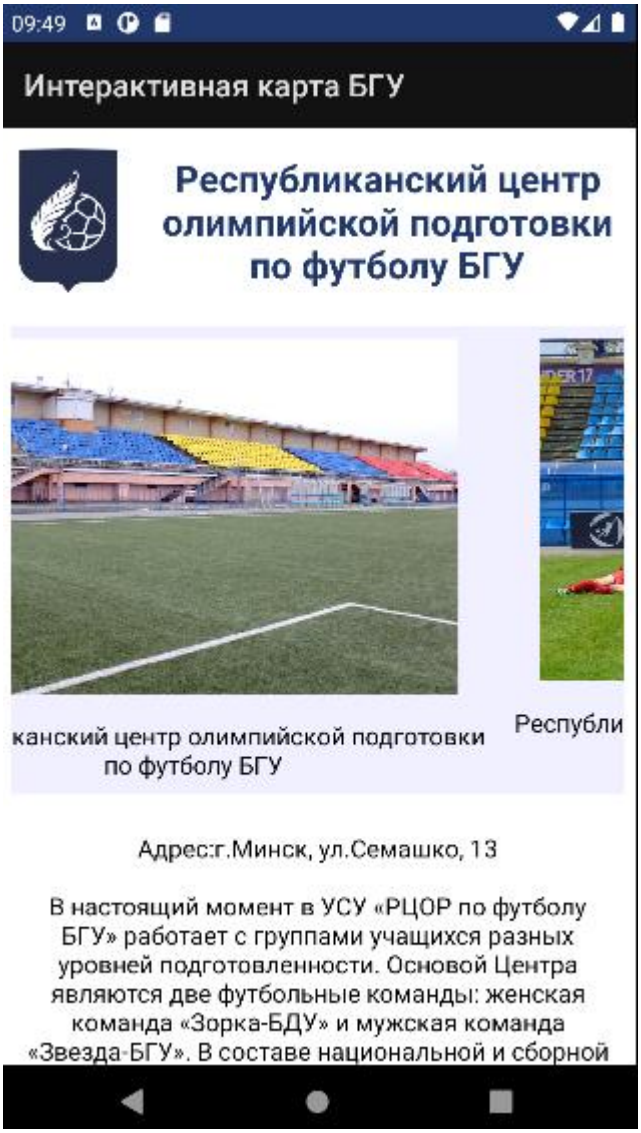

Рисунок 2.9 – Фрагмент с информацией о современном объекте

Таким образом, была совершена реализация открытия фрагментов с информацией и галереей изображений об исторических и современных объектах при клике пользователя на соответствующую кнопку.

ЗАКЛЮЧЕНИЕ

В результате проделанной работы удалось реализовать мобильное приложение под операционную систему Android, которое работает с технологией Mapbox и отображает информацию о достопримечательностях БГУ.

Данное приложение включает в себя ряд технологий, используемых Android-разработчиками, включая API Mapbox. В частности, в приложении происходит совместная работа с базой данных, внедрением зависимостей, картами и другими библиотеками.

С помощью текущей версии приложения пользователь может узнать информацию о местоположении различных достопримечательностей (как исторических, так и современных), а также узнать информацию о них при помощи клика по маркеру на карте. Более того, для каждого из объектов пользователь имеет возможность открыть отдельный фрагмент с более подробной информацией, а также галереей из изображений, посвящённых этим объектам.

Стоит отметить, что хранение данных в приложении уже реализовано на разных уровнях абстракции. В приложении есть как доменные сущности, так и сущности базы данных. Это позволит облегчить модификацию приложения в будущем, так как происходит разделение структуры сущностей от деталей их реализации.

За счёт использования локальной базы данных, можно кешировать данные на мобильном устройстве пользователя, благодаря чему улучшается производительность приложения. Помимо этого, происходит регулярное обновление данных в фоновом режиме.

В качестве возможных модификаций для данного проекта можно отметить добавление в проект функции построения маршрутов для пользователя, чтобы указывать пользователю путь и, тем самым, облегчать ознакомление с достопримечательностями. Помимо маршрутов, также имеет место быть добавление нескольких слоёв для отображения достопримечательностей в зависимости от временной шкалы (по годам). Данная функция может быть очень полезна тем, кто желает изучить историю БГУ с помощью данного приложения. Также, в приложении есть потенциал к добавлению режима просмотра достопримечательностей в 3D формате.

Работа над данным проектом длилась около девяти месяцев. За это время проект обрёл ряд функций и возможностей. Данный проект может быть полезен студентам и преподавателям, желающим получше ознакомиться с достопримечательностями БГУ, а также туристам, которые желают ознакомиться с историей Беларуси в целом.