образование ПОЛИТЕХ одноклассники экосистема

Работа с сетью

Нуртдинов Артур

План лекции

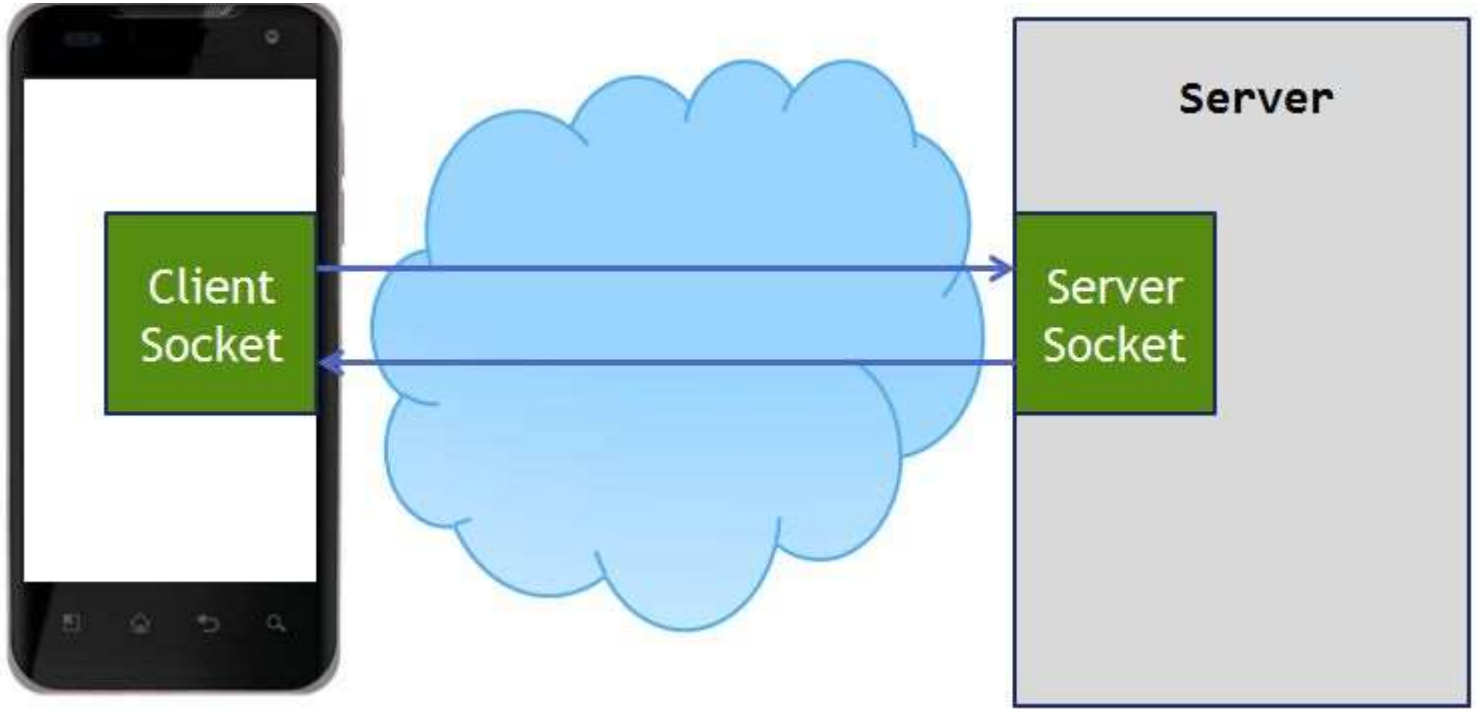
1. Особенности работы с сетью в мобильных приложениях
2. Подходы по реализации сетевого взаимодействия
3. Выполнение https запросов
4. Парсинг ответа с сервера

Особенности работы с сетью в мобильных приложениях

1. Трафик – это деньги пользователя. Не всегда есть возможность использовать Wi-Fi, а значит нужно следить за сетью пользователя прежде чем скачивать большие файлы или выполнять большие запросы данных, например для синхронизации кэша.
2. Лимит батарейки. Выполнение сетевых запросов – одно из основных энергопотребляющих действий, не стоит злоупотреблять этим.
3. Безопасность. Сетевое соединение нужно делать защищённым. Помимо этого, если у пользователя есть возможность загружать файлы, необходимо защищать других пользователей от запрещённого контента и небезопасных файлов и вовремя удалять любую запрещенную информацию.
4. Медленное соединение, смена сети, кратковременные пропажи сети (например в лифте). Такие ситуации необходимо корректно обрабатывать и показывать пользователю информативное сообщение или состояние экрана (например отображать лоадер, попытки соединения и тд).
5. NetworkOnMainThreadException – это Runtime exception, который выбрасывается, если ваше приложение пытается совершить сетевую операцию на главном (UI) потоке.

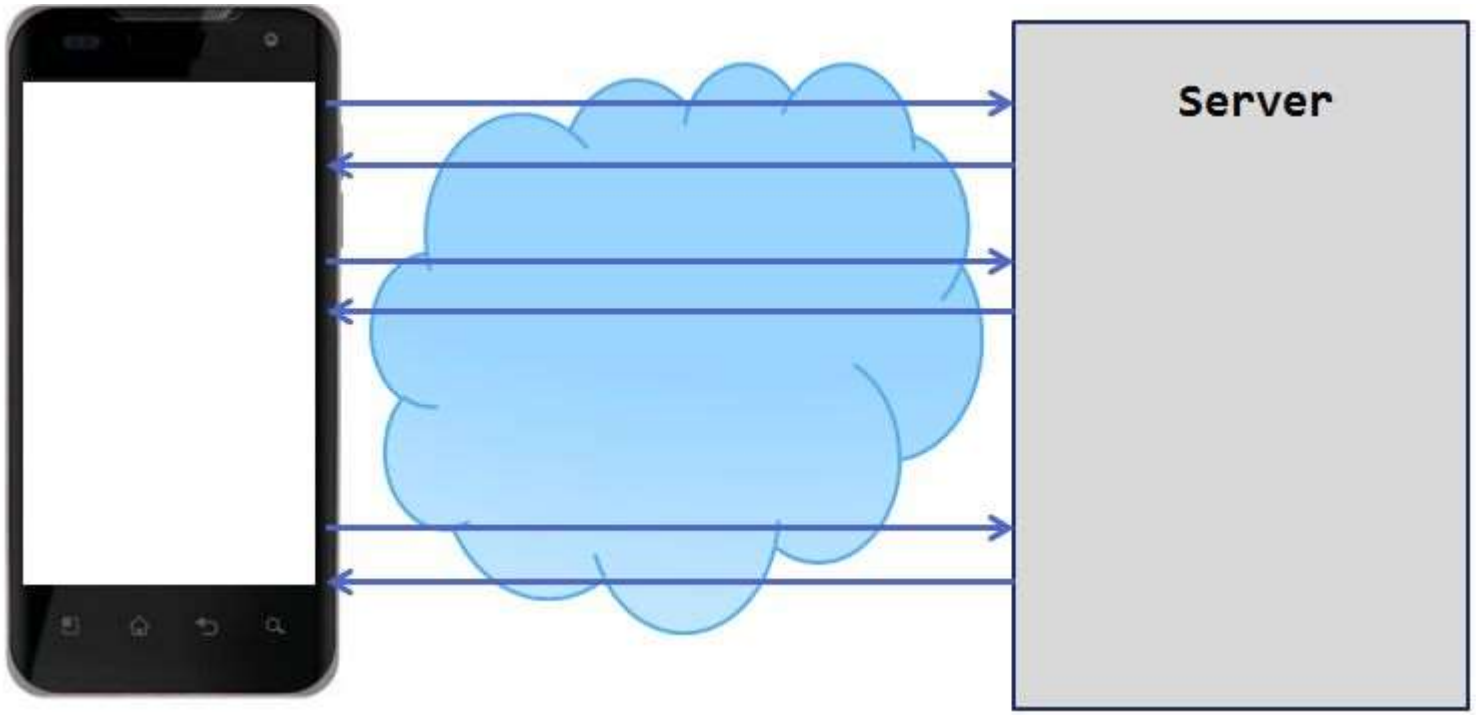
Подходы по реализации сетевого взаимодействия

1. **SocketApi**. Используется, когда важна скорость доставки сообщения, важен порядок доставки сообщений и необходимо держать стабильное соединение с сервером. Такой способ зачастую реализуется в мессенджерах и играх.



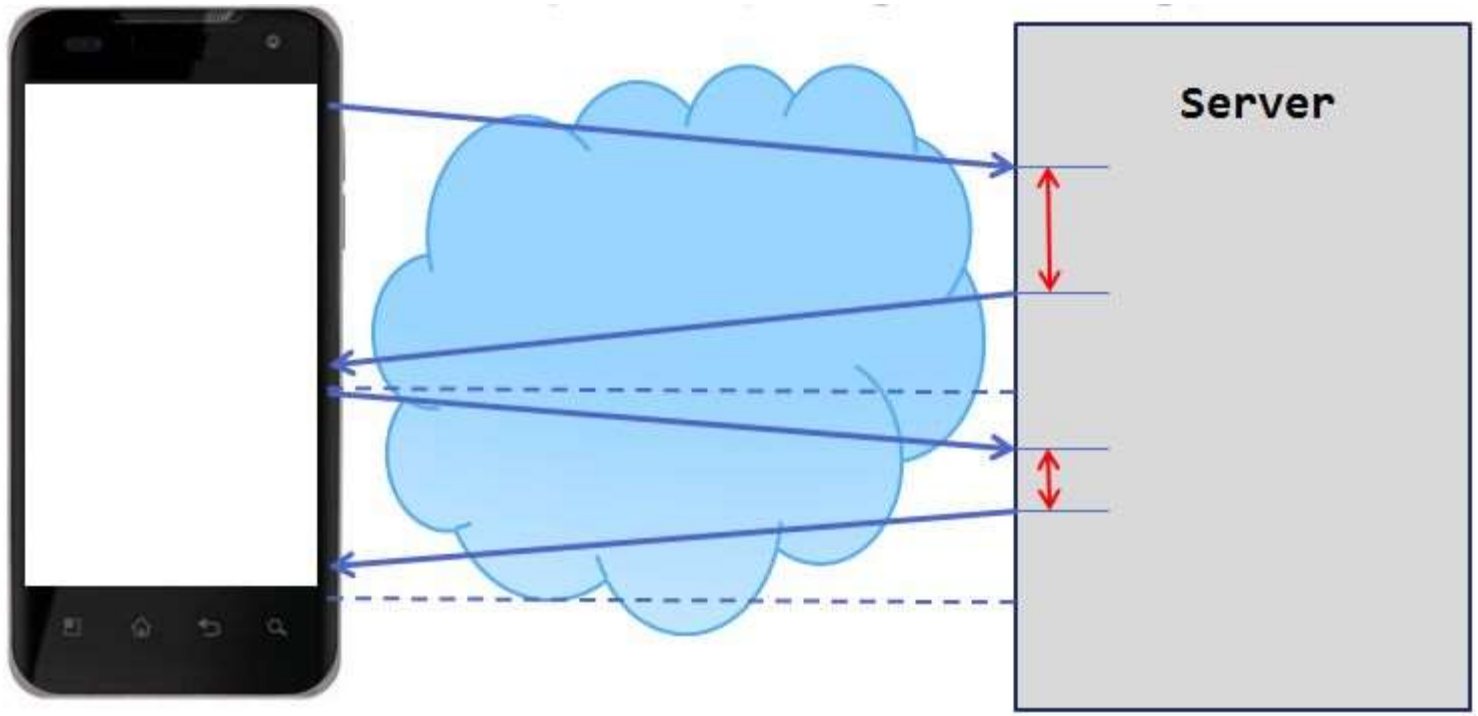
Подходы по реализации сетевого взаимодействия

2. **Polling.** Клиент посылает запрос на сервер и говорит ему: «Дай мне свежие данные»; сервер отвечает на запрос клиента и отдает все, что у него накопилось к этому моменту. Минус такого подхода в том, что клиент не знает, появились ли свежие данные на сервере. По сети лишний раз гоняется трафик, в первую очередь из-за частых установок соединений с сервером.



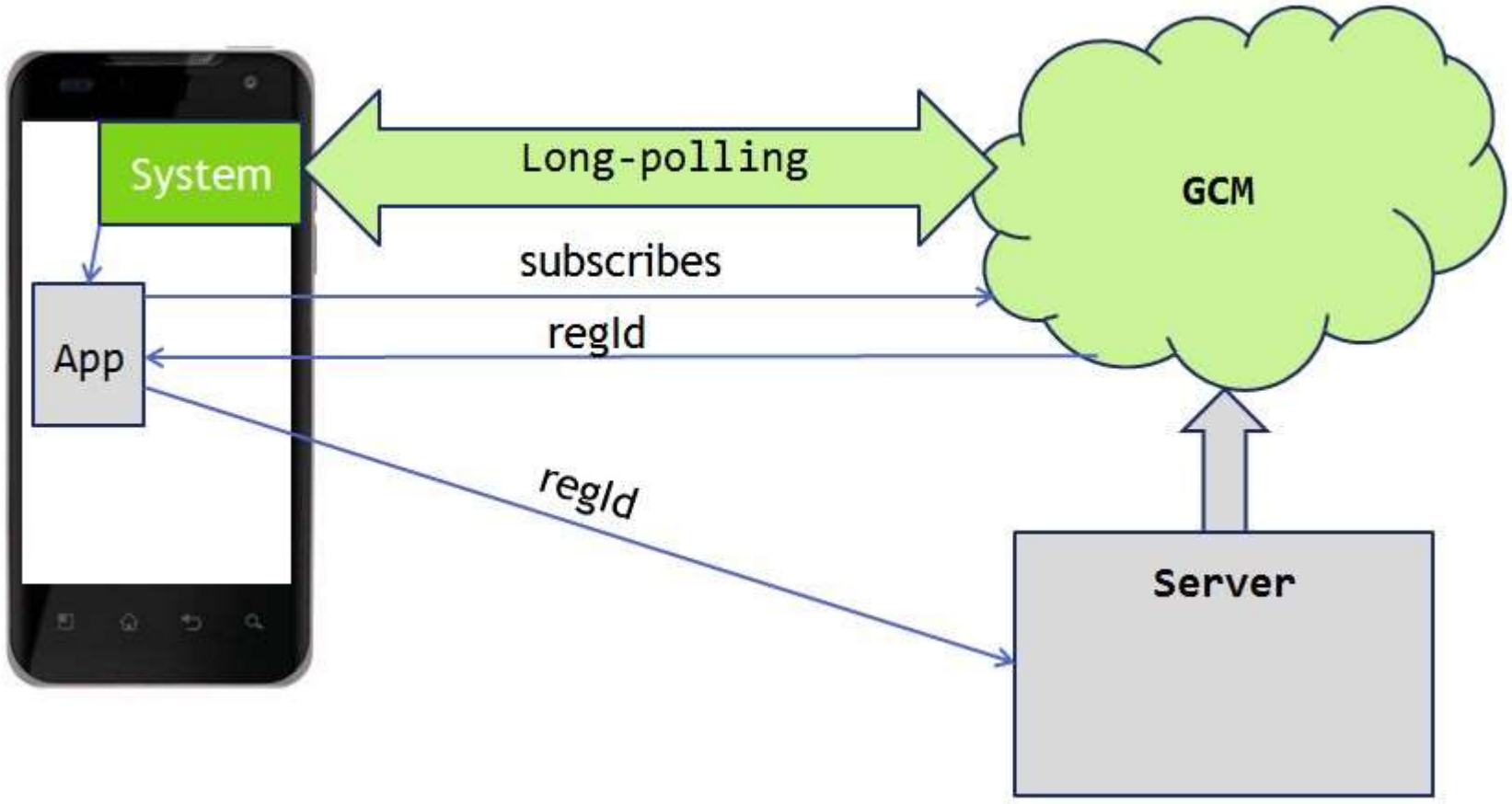
Подходы по реализации сетевого взаимодействия

3. **Long polling**. Клиент посылает «ожидающий» запрос на сервер. Сервер смотрит, есть ли свежие данные для клиента, если их нет, то он держит соединение с клиентом до тех пор, пока эти данные не появятся. Как только данные появились, он «пушит» их обратно клиенту. Клиент, получив данные от сервера, тут же посылает следующий «ожидающий» запрос и т.д.



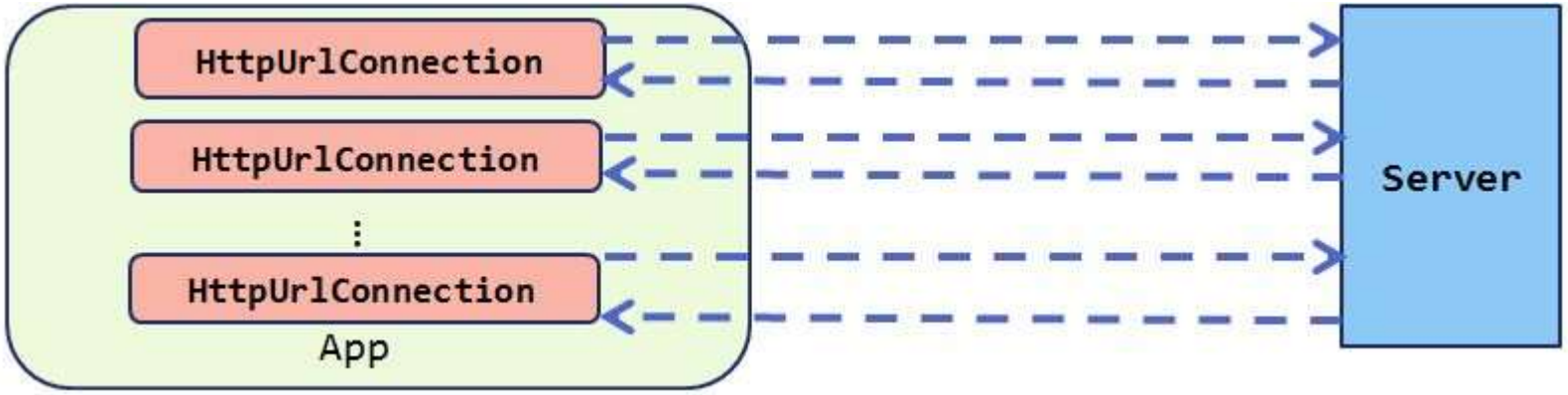
Подходы по реализации сетевого взаимодействия

4. **Google Cloud Messaging**. Механизм получения пуш-уведомлений с вашего сервера через облако GCM. Механизм long polling, или пуш-уведомлений (push notifications), реализован в самой платформе Android.



Выполнение https запросов

1. **HttpUrlConnection**. В случае HttpUrlConnection следует создавать на каждый запрос новый экземпляр клиента.




```
URL url = new URL("http://www.android.com/");
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
try {
  InputStream in = new BufferedInputStream(urlConnection.getInputStream());
  readStream(in);
} finally {
  urlConnection.disconnect();
}
```

Выполнение https запросов

1. **HttpUrlConnection**. Отправка запроса на сервер

```
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
try {
  urlConnection.setDoOutput(true);
  urlConnection.setChunkedStreamingMode(0);

  OutputStream out = new BufferedOutputStream(urlConnection.getOutputStream());
  writeStream(out);

  InputStream in = new BufferedInputStream(urlConnection.getInputStream());
  readStream(in);
} finally {
  urlConnection.disconnect();
}
```

Выполнение https запросов

2. Использование Retrofit + OkHttp + Gson.

Retrofit – библиотека для сетевого взаимодействия на основе паттерна Каскад. Интегрируется с другими как сетевыми библиотеками, так и реактивными

OkHttp - простой, легкий в использовании API для выполнения HTTP-запросов, включая поддержку протоколов HTTP/1.1 и HTTP/2.

OkHttp — это библиотека более низкого уровня, чем Retrofit. Это означает, что HTTP-запросы, автоматизированные в Retrofit с помощью аннотаций, придётся писать вручную. Однако в этом и главный плюс библиотеки: она предоставляет более обширный функционал и настройки соединения, что может повысить производительность и сократить использование памяти. К слову, Retrofit под капотом использует OkHttp.

Gson позволяет конвертировать объекты JSON в Java-объекты и наоборот.

образование ПОЛИТЕХ одноклассники экосистема

Retrofit. Подключение зависимостей

```
private val gson by lazy { "com.google.code.gson:gson:${Versions.gson}" }
private val okhttp by lazy { "com.squareup.okhttp3:okhttp:${Versions.okhttp}" }
private val retrofit by lazy { "com.squareup.retrofit2:retrofit:${Versions.retrofit}" }
private val gsonConverter by lazy { "com.squareup.retrofit2:converter-gson:${Versions.retrofit}" }
```

Retrofit. Интерфейс Api

```
public interface Api {
    @POST("/auth/signin/google")
    suspend fun authGoogle(@Body codeBody: GoogleAuthCodeBody): Response<TokensResponseBody>

    @POST("/auth/logout")
    suspend fun logout(@Body refreshTokenBody: RefreshTokenBody): Response<Void>

    @HTTP(method = "DELETE", path = "/events/{id}", hasBody = true)
    suspend fun deleteEvent(
        @Path("id") id: String,
        @Body onlyDeleteInstanceBody: OnlyDeleteInstanceBody
    ): Response<Void>


    @GET("/user")
    suspend fun getUser(): Response<UserResponse>

    @GET("/users")
    suspend fun searchUsers(
        @Query("query") query: String,
        @Query("limit") limit: Int,
        @Query("page") page: Int
    ): Response<SearchResponse>
```

Retrofit. POJO классы для использования в Api

```kotlin
data class GroupBody(
    @SerializedName("name")
    val name: String,
    @SerializedName("color")
    val color: String,
    @SerializedName("users_ids")
    val usersIds: List<Long>,
)
```

```kotlin
data class GroupByIdResponse(
    @SerializedName("id")
    val id: Long,
    @SerializedName("name")
    val name: String,
    @SerializedName("color")
    val color: String,
    @SerializedName("creator_id")
    val creatorId: Long,
    @SerializedName("users")
    val users: List<UserResponse>
)
```

```kotlin
data class UserResponse(
    @SerializedName("id")
    val id: Long,
    @SerializedName("full_name")
    val fullName: String,
    @SerializedName("email")
    val email: String?,
    @SerializedName("phone_number")
    val phone: String?,
    @SerializedName("photo")
    val photo: String?,
)
```

Указываем это в proguard-rules.pro:

```
# Network region
-keepclassmembers class ru.spbstu.common.network.model.* {
  *;
}
# End Network region
```

Retrofit. Создание реализации Api

```
@Provides
@ApplicationScope
fun provideRetrofit(client: OkHttpClient, gson: Gson): Retrofit =
    Retrofit.Builder() Retrofit.Builder
        .baseUrl(BuildConfig.ENDPOINT) Retrofit.Builder
        .client(client)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build()
```

```
@Provides
@ApplicationScope
fun provideApi(retrofit: Retrofit): Api = retrofit.create(Api::class.java)
```

Retrofit. Gson

```
@Provides
@ApplicationScope
fun provideGson(): Gson = GsonBuilder().create()
```

Всё, что касается сериализации JSON строк в POJO классы настраивается с помощью GsonBuilder()
Например некоторые методы:

setLenient() – По умолчанию Gson принимает только строки JSON спецификации **RFC 4627**

setDateFormat(String pattern) – Для сериализации Date объектов по заданному паттерну

serializeNulls() – Для сериализации null полей. Gson не будет пропускать null поля при записи их в JSON формат

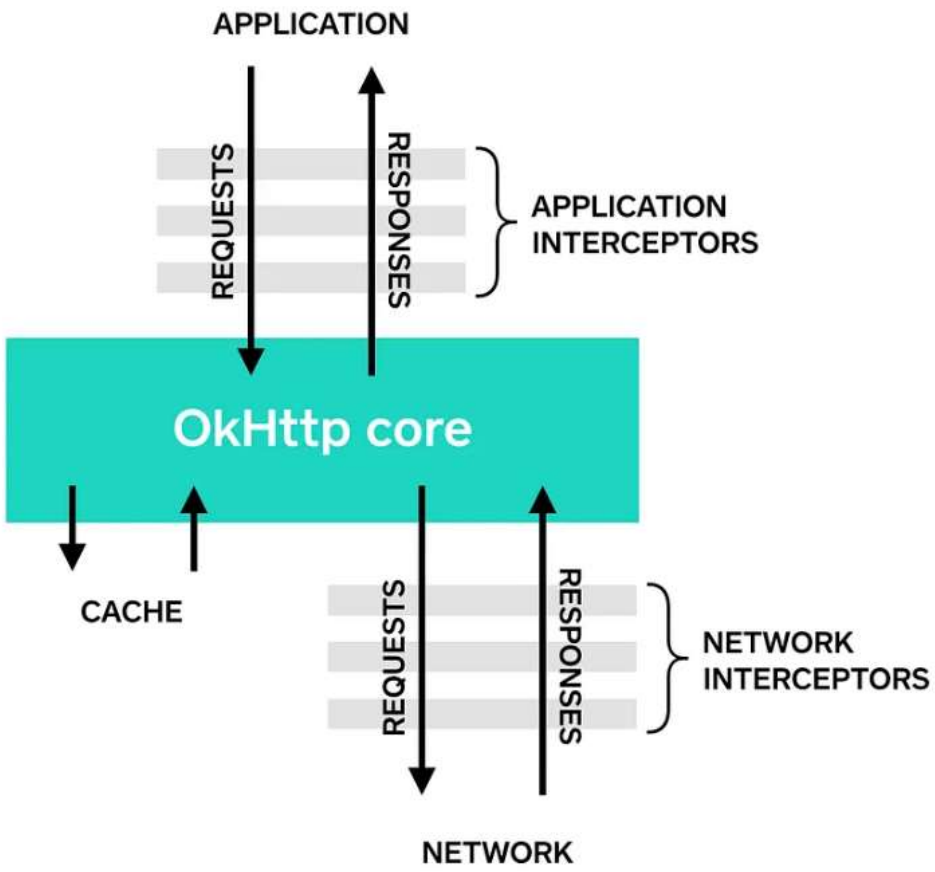
disableHtmlEscaping() – Для настройки Gson, чтобы не пропускать html символы, такие как < >

Retrofit. OkHttpClient

```kotlin
@Provides
@ApplicationScope
fun provideOkHttpClient(
    restInterceptor: Interceptor,
): OkHttpClient {
    val builder = OkHttpClient.Builder()
        .readTimeout( timeout: 10, TimeUnit.SECONDS)
        .connectTimeout( timeout: 10, TimeUnit.SECONDS)
        .callTimeout( timeout: 10, TimeUnit.SECONDS)
        .addInterceptor(restInterceptor)
    return builder.build()
}
```

Builder - класс предоставляющий методы для настройки клиента, например кэш, аутентификация, перехватчики, тайм-ауты и др. По завершению настройки используется метод build(), который возвращает экземпляр класса OkHttpClient.
Перехватчики (Interceptor) часто используются для того, чтобы подсунуть токен авторизации в запрос, чтобы не делать это к каждому запросу вручную или для обновления токена авторизации при его истечении

Retrofit. OkHttpClient Interceptor



Network Interceptors работают на сетевом уровне и являются хорошим местом для написания логики, не зависящей от данных запроса (например ретраев)

Application Interceptors работают на уровне вашего Application и являются хорошим место для написания логики, зависящей от данных запроса (например подсовывания токена авторизации)

Retrofit. OkHttpClient Interceptor

```kotlin
@Provides
@ApplicationScope
internal fun provideRestInterceptor(
    gson: Gson,
    preferencesRepository: PreferencesRepository,
    @Named("refresh") refreshOkhttpClient: OkHttpClient,
): Interceptor =
    Interceptor { chain ->
        val original = chain.request()
        val accessToken = preferencesRepository.token
        val requestBuilder = original.newBuilder()
        if (accessToken.isNotEmpty() && !original.url.toString().contains( other: "auth")) {
            requestBuilder.addHeader(AUTHORIZATION, value: BEARER + accessToken)
        }
        val request = requestBuilder.build()
        return@Interceptor chain.proceed(request)
    }
```

Retrofit. Authenticator

Плюсы использования Authenticator:

- Автоматический ретрай (по дефолту – до 20 раз)
- Не нужно проверять ответ
- Нет сайд-эффектов когда используются несколько интерсепторов

Authenticator вызывается каждый раз, когда ваш сервер отвечает кодом 401 (UNAUTHORIZED)

```
override fun authenticate(route: Route?, response: Response): Request? {
    // We need to have a token in order to refresh it.
    val token = tokensRepository.token ?: return null
    synchronized( lock: this) {
        val newToken = tokensRepository.token
        // Check if the request made was previously made as an authenticated request.
        if (response.request.header( name: "Authorization") != null) {
            // If the token has changed since the request was made, use the new token.
            if (newToken != token) {
                return response.request Request
                    .newBuilder() Request.Builder
                    .removeHeader( name: "Authorization")
                    .addHeader( name: "Authorization",  value: "Bearer $newToken")
                    .build()
            }
            val updatedToken = tokensRepository.refreshToken() ?: return null
            // Retry the request with the new token.
            return response.request Request
                .newBuilder() Request.Builder
                .removeHeader( name: "Authorization")
                .addHeader( name: "Authorization",  value: "Bearer $updatedToken")
                .build()
        }
    }
    return null
}
```

Материалы

1. https://habr.com/ru/companies/vk/articles/185696/ - Работа с сетью в Android: трафик, безопасность и батарейка
2. https://developer.android.com/reference/java/net/HttpURLConnection - HttpUrlConnection
3. https://github.com/square/retrofit/blob/master/retrofit/src/main/resources/META-INF/proguard/retrofit2.pro - настройки proguard для Retrofit

Спасибо!

образование

ПОЛИТЕХ

одноклассники
экосистема