6.3. Передача данных по сети. Часть 2

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 6.3. Передача данных по сети. Часть 2

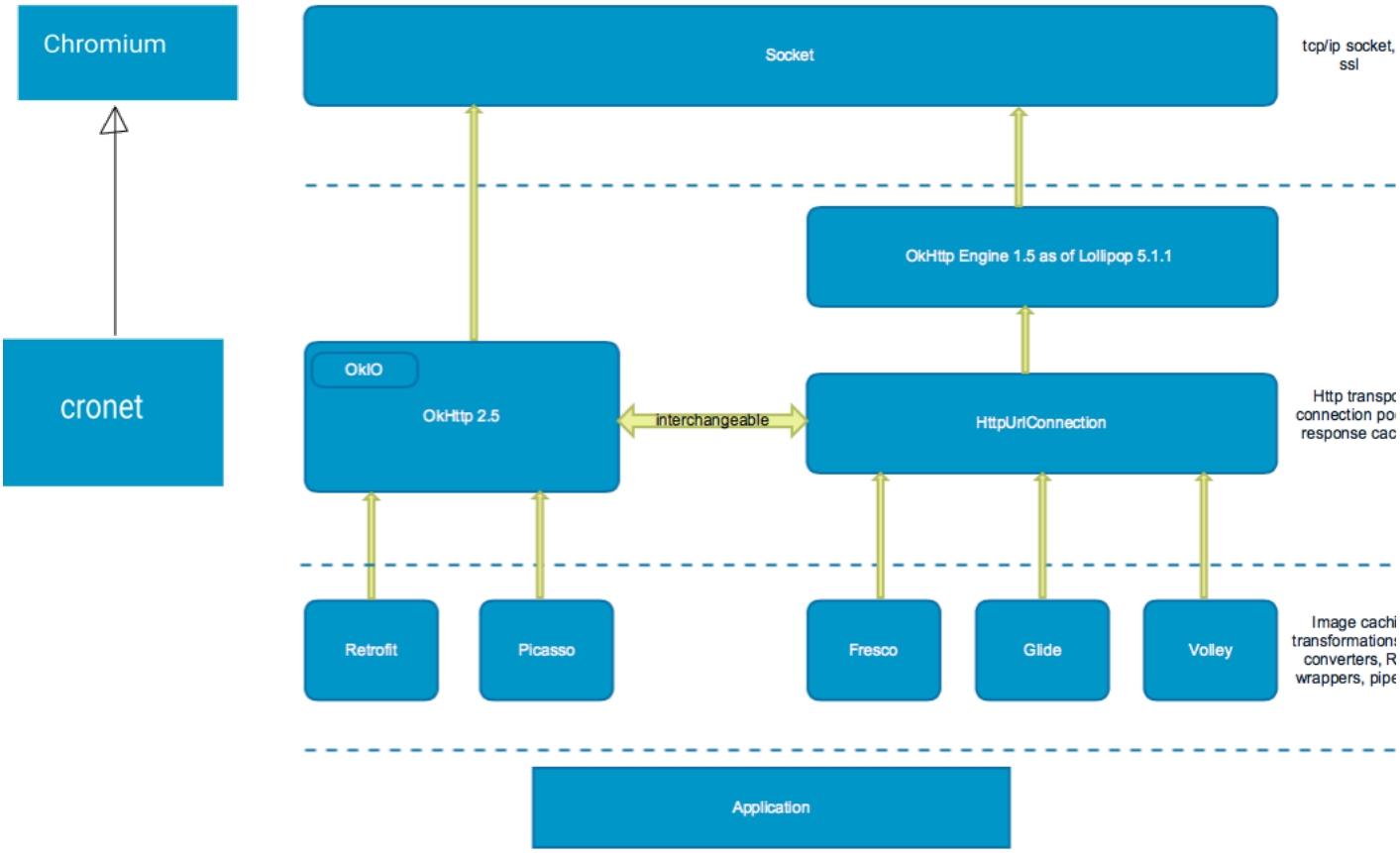
Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 19:15

Оглавление

1. 6.3.1 Сетевые библиотеки Android

Для использования клиент-серверной модели разработано множество разнообразных библиотек для Android которые могут помочь разработчику в создании своего приложения. Перечислим наиболее известные на сегодня:

- AndroidHttpClient - библиотека уровня HTTP основанная на коде проекта Appache, на сегодня устарела.
- OkHttp - OkHTTP - это сетевая библиотека HTTP-клиента. Одна предназначена для обмена данными и другими формами мультимедиа с онлайн-серверами путем отправки запросов. Используется и как самостоятельная библиотека на уровне HTTP, так и как базовая для других, более высокоуровневых библиотек.
- Fast Android Networking - это расширенная версия библиотеки OkHTTP. Она построена поверх OkHttp. Это быстрая и эффективная библиотека с добавлением новых функций.
- Voley - это сетевая HTTP-библиотека для Android, которая позволяет разработчикам быстро и просто отправлять запросы на серверы из приложения. Позволяет управлять конкурентными запросами, получение данных синхронно и асинхронно, кеширование запросов и т.д.
- Cronet - сетевая библиотека для приложений Android и iOS. Cronet - основан на стеке Chromium Network. Cronet может улучшить сетевую производительность приложения за счет уменьшения задержки и увеличения пропускной способности.
- Retrofit - это типобезопасный HTTP-клиент для Android и Java. Кроме хорошо реализованной широкой функциональности по работе в клиент-сереврвеной архитектуре, дополнительно встроена поддержка сериализации и REST взаимодействия.



Все рассмотренные библиотеки могут обеспечить взаимодействие с сервером посредством HTTP запросов. Однако они отличаются предоставляемым объемом функциональности, а также дополнительными возможностями.

2. 6.3.2 Библиотека Retrofit

Рассмотрим одну из популярных библиотек - Retrofit. К ее достоинствам относят:

- нет нужды делать запросы к HTTP API в отдельном потоке;
- сокращается длина кода и, соответственно, ускоряется разработка;
- возможно подключение стандартных пакетов для конвертации JSON в объекты и обратно (например, пакета Gson);
- динамическое построение запросов;
- обработка ошибок;
- упрощенная передача файлов.

Рассматриваемые далее примеры входят в практическую работу, исходный код которой можно скачать по ссылке

Рассмотрим пример клиент-серверного приложения для Android на основе библиотеки Retrofit2. Пример будет похож на пример из прошлой лекции, но с расширенным функционалом. Особенно интересно, что внутри приложения для передачи на сервер и обратно будет использоваться не только строки но и объекты собственного класса **User** или даже их списки:

```
public class User {
    public String firstName;
    public String lastName;
}
```

Конечно передаются не сами объекты, а все таки строки полученные путем сериализации в JSON. Фреймворки Retrofit и Spring осуществляют прозрачную сериализацию/десериализацию любых объектов, классы которых присутствуют и на сервере, и на клиенте. Но в начале, для использования библиотеки необходимо добавить следующие записи в файл **build.gradle** (app):

```
    implementation 'com.google.code.gson:gson:2.8.5'
    implementation 'com.squareup.retrofit2:retrofit:2.4.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.4.0'
```

А также в манифест добавить разрешение для работы с сетью и разрешение устанавливать соединение по нешифрованному протоколу HTTP:

```
  <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:usesCleartextTraffic="true"
```

После перечисленных действий можно приступать к написанию кода сетевого взаимодействия с использованием Retrofit. Логика работы библиотеки основывается на аннотациях. Благодаря их использованию можно описывать запросы на сервер. Для описания запросов к серверу необходимо объявить интерфейс, который впоследствии будет использоваться при генерации запросов. Перед каждым методом интерфейса должна стоять аннотация, основываясь на которой, Retrofit определяет, какого типа запрос обрабатывается данным методом. Также с помощью аннотаций можно указывать параметры запроса. Вот так, например, выглядит описание GET и POST запросов:

```
interface UserController {

   @POST("/hello")
    fun hello(@Body user: User): Call<Boolean>

    @GET("/list")
    fun list(): Call<List<User>>

    // ...
}
```

Важно отметить, что адрес сайта, к которому отправляется запрос, не указывается. Он будет передан на этапе создания соединения клиент-сервер. Аннотация содержит лишь путь к ресурсу на сервере - URI. Во фреймворке Retrofit для описания HTTP методов, также есть аннотации GET, POST, PUT, PATCH, DELETE, OPTIONS и HEAD.

Можно также использовать динамический маппинг URI:

```
    @GET("/get/Murad/Rezvan")
    fun get(@Path("firstName") firstName:String , @Path("lastName") lastName: String ): Call<User>
```

В этом случае Retrofit заменит Murad и Rezvan на фактические части URL, которые и будут переданы методу при вызове. В этом случае аргументы метода должен быть аннотированы **@Path**, а в скобках должно заключаться обозначение конкретной части URI. В общем же случае, чтобы задать задать запросу параметры, в Retrofit используют аннотации:

- **@Query** - передача параметров в URL после символа "?", обычно применяется в GET запросе,

- **@Path** - передача параметров в URI как часть пути, может применяться в любых запросах,
- **@Body** - передача одного параметра в теле запроса, обычно применяется в PUT запросе,
- и другие

Создание соединения клиент-сервер выглядит, как серия вызовов методов библиотеки Retrofit. Например, так:

```
val retrofit = Retrofit.Builder()
            .baseUrl(connectionURL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
val service: UserController = retrofit.create(UserController::class.java)
```

Через параметр метода **baseUrl()** передается адрес сервера. **GsonConverterFactory** — это класс для конвертации объектов в JSON. В нем используется уже рассмотренный нами класс **Gson**. Что важно, Retrofut самостоятельно создаст объект-реализацию интерфейса **UserController** — с функциями выполняющими реальные запросы HTTP API.

После того, как соединение создано, можно его эксплуатировать, вызывая методы, привязанные к HTTP API:

```
val call: Call<User> = service.get(firstname.text.toString(), lastname.text.toString())
```

В приведенном примере, из объекта созданного Retrofit-ом на основании объявленного выше интерфейса, вызывается интересующий метод - в данном примере **get()** со строковыми параметрами. Результатом этого вызова является объект типа **Call** , параметризованный типом, объект которого ожидается в ответе от сервера. Далее из полученного объекта можно вызвать методы:

- **execute()** - для синхронного вызова, т.е. выполнение приложения будет приостановлено, до получения ответа от сервера. В этом случае вызов нужно осуществлять в потоке.
- **enqueue()** - для асинхронного вызова. В этом случае выполнение приложение продолжится, а по факту получения ответа от сервера, сработает вызов метода **onResponse()** или **onFailure()** зарегистрированного слушателя (объект **Callback**). В этом случае выполнять в потоке вызовы не нужно.

Пример синхронного вызова (обработчик кнопки send):

```
send.setOnClickListener(View.OnClickListener {
            Thread() {
                run {
                    val call: Call<Boolean> = service.hello(
                        User(firstname.text.toString(), lastname.text.toString())
                    )
                    val userResponse:Response<Boolean>  = call.execute()
                    val result: Boolean? = userResponse.body()
                    Log.d("RetrofitClient","Send data to server was: "+result)
                }
            }.start()
        })
```

Как видно, сначала создается вызов запроса, и лишь потом он выполняется методом execute(). Следует обратить внимание, что если попытаться выполнить вызов запроса дважды, то будет брошено исключение IllegalStateException.

Пример асинхронного вызова:

```
 get.setOnClickListener(View.OnClickListener {
            result.text=""
            val call: Call<User> =  service.get(firstname.text.toString(), lastname.text.toString())
            call.enqueue(object : Callback<User> {
                override fun onResponse(call: Call<User>, response: Response<User>) {
                    if (response.isSuccessful) {
                        val user: User? = response.body()
                        result.text = "найден: " + user?.firstName + " " + user?.lastName
                    }
                }
                override fun onFailure(call: Call<User>, t: Throwable) {
                    Log.d("RetrofitClient","Receive data from server problem ")
                }
            })
        })
```

В этом примере видно, что вместо метода **execute()** , вызывается **enqueue()** в который передается объект колбэка, из которого, при успешном получении ответа от сервера, будет вызван метод **onResponse()**

> Дополнительную информацию по фреймворку Retrofit можно увидеть на сайте проекта.

3. 6.3.3 Пример реализации

Модифицируем программу из примера 6.2 так, чтобы она использовала JSON и библиотеку Retrofit. Полученный код клиентской части приведен ниже.

Исходный код примера можно скачать по ссылке

В проектируемой программе необходимо реализовать три сценария взаимодействия:

1. Используя запрос /hello серверного веб API, необходимо передать на сервер объект User. Сервер, в случае его успешного внесения в список пользователей, должен вернуть true.
2. Используя запрос /get серверного веб API, необходимо передать на сервер строки с именем и фамилией пользователя. Сервер, в случае их нахождения в списке пользователей, должен вернуть объект найденного пользователя.
3. Используя запрос /list серверного веб API, необходимо необходимо получить от сервера список объектов пользователей.

Рассмотрение кода проекта начнем с класс передаваемой структуры данных User.kt.

```
class User (var firstName:String ,var lastName: String )
```

На стороне сервера этот же класс описан идентичным образом. Обратите внимание, что для таких сериализуемых объектов правилом хорошего тона является наличие акссессоров, а также конструктора по умолчанию. В минимальном варианте он может выглядеть следующим образом:

```
public class User {
    public String firstName;
    public String lastName;
    public User(String firstName,String lastName) {
        this.firstName=firstName;
        this.lastName=lastName;
    }
}
```

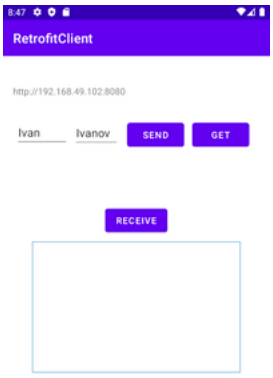
Интерфейс клиент-серверного взаимодействия UserController.kt :

```
interface UserController {
    @POST("/hello")
    fun hello(@Body user: User): Call<Boolean>

    @GET("/list")
    fun list(): Call<List<User>>

    @GET("/get/Murad/Rezvan")
    fun get(@Path("firstName") firstName:String , @Path("lastName") lastName: String ): Call<User>
}
```

Код активности следующий:

```
class MainActivity : AppCompatActivity() {
    val connectionURL: String = "http://192.168.49.102:8080"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val et_url: TextView = findViewById(R.id.url)
        et_url.text = connectionURL;
        val firstname: EditText = findViewById(R.id.firstname)
        val lastname: EditText = findViewById(R.id.lastname)
        val send: Button = findViewById(R.id.send)
        val get: Button = findViewById(R.id.get)
        val receive: Button = findViewById(R.id.receive)
        val result: TextView = findViewById(R.id.result)
        val retrofit = Retrofit.Builder()
            .baseUrl(connectionURL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
        val service: UserController = retrofit.create(UserController::class.java)
        send.setOnClickListener(View.OnClickListener {
            Thread() {
                run {
                    val call: Call<Boolean> = service.hello(
                        User(firstname.text.toString(), lastname.text.toString())
                    )
                    val userResponse:Response<Boolean>  = call.execute()
                    val result: Boolean? = userResponse.body()
                    Log.d("RetrofitClient","Send data to server was: "+result)
                }
            }.start()
        })
        receive.setOnClickListener(View.OnClickListener {
            result.text=""
            val call: Call<List<User>> = service.list()
            call.enqueue(object : Callback<List<User>> {
                override fun onResponse(call: Call<List<User>>, response: Response<List<User>>) {
                    if (response.isSuccessful) {
                        val list: List<User>? = response.body()
                        var strList: String = ""
                        list?.forEach { strList += it.firstName + "  " + it.lastName + "\n" }
                        result.text = strList
                    }
                }
                override fun onFailure(call: Call<List<User>>, t: Throwable) {
                    Log.d("RetrofitClient","Receive data from server problem ")
                }
            })
        })
        get.setOnClickListener(View.OnClickListener {
            result.text=""
            val call: Call<User> =
                service.get(firstname.text.toString(), lastname.text.toString())
            call.enqueue(object : Callback<User> {
                override fun onResponse(call: Call<User>, response: Response<User>) {
                    if (response.isSuccessful) {
                        val user: User? = response.body()
                        result.text = "найден: " + user?.firstName + " " + user?.lastName
                    }
                }
                override fun onFailure(call: Call<User>, t: Throwable) {
                    Log.d("RetrofitClient","Receive data from server problem ")
                }
            })
        })
    }
}
```

Рабочий клиент будет выглядеть примерно следующим образом:



Рассмотрим серверную часть программы.

```
@RestController
public class UserController {

    List<User> list=new LinkedList<User>();

    @RequestMapping(method = RequestMethod.GET,value = "/hello")
    public String hello(@RequestParam String firstName, @RequestParam String lastName) {
        list.add(new User(firstName,lastName));
        System.out.println(firstName);
        String rez= "Здравствуйте " + firstName.substring(0,1)+"."+lastName +"!";
        return rez;
    }

    @RequestMapping(method = RequestMethod.POST,value = "/hello")
    public Boolean hello(@RequestBody User user) {
        System.out.println(user.firstName);
        list.add(user);
        return new Boolean(true);
    }

    @RequestMapping("/list")
    public List<User> list(){
        return list;
    }

    @RequestMapping("/get/Murad/Rezvan")
    public User get(@PathVariable String firstName,@PathVariable String lastName){
        for(User user:list) {
            if(user.firstName.equals(firstName) && user.lastName.endsWith(lastName))
                return user;
        }
        return null;
    }

}
```

В этом примере первый метод **hello()** оставлен для совместимости с примером из предыдущей лекции, а остальные три метода соответствуют описанному в постановке задаче веб API.

Начать тур для пользователя на этой странице