

Android: a full-stack to consume a REST API

Romain Rochegude

2016.09.30

Introduction

Introduction

- Well-known concerns:
 - Communicate with remote API
 - Parse content and deal with it
 - Do it asynchronously
 - Notify components of job termination

REST client: Retrofit

REST client: Retrofit

- Well-known, documented, “must-have” Android library
- Write a Java interface to declare API method
- Annotations to describe the HTTP request
 - HTTP method (@GET, @POST, etc.)
 - URL parameter (@Path)
 - query parameter (@Query, @QueryMap)
 - request body (@Body)
 - multipart request body (@Multipart, @Part)
 - form management (@FormUrlEncoded, @Field)
 - headers management (@Headers)

```
public interface GitHubService {  
  
    @GET("/users/{user}/repos")  
    Call<List<DTORepo>> listRepos(  
        @Path("user") String user);  
  
}
```

- Build at runtime an implementation

```
Retrofit retrofit = new  
    Retrofit.Builder()  
        .baseUrl("https://api.github.com")  
        .build();  
  
GitHubService service =  
    retrofit.create(GitHubService.class);
```

- Simple calls

```
Call<List<DTORepo>> repos =  
    service.listRepos("RoRoche");
```

- Converters to (de)serialize HTTP bodies

```
Retrofit retrofit = new  
    Retrofit.Builder()  
        //...  
        .addConverterFactory(  
            GsonConverterFactory.create())  
        .build();
```


- To add a new HTTP request:
 - declare DTO class(es) with your parsing strategy
 - declare body class (optional)
 - declare the method in Java interface with suitable annotations
- Minimum amount of code to deal with remote API

JSON parser: LoganSquare

JSON parser: LoganSquare

- Faster, according to BlueLine Labs benchmark
- Clear annotations

```
@JsonObject
public class DTORepo {
    @JsonField(name = "id")
    public Integer id;

    @JsonField(name = "name")
    public String name;
}
```

- Available retrofit converter

```
Retrofit retrofit = new  
    Retrofit.Builder()  
        //...  
        .addConverterFactory(  
            LoganSquareConverterFactory.create()  
        ).build();
```

- Simple parsing methods

// Parse from an InputStream

```
InputStream is = //...
```

```
Image image = LoganSquare.parse(is,  
    Image.class);
```

// Parse from a String

```
String json = //...
```

```
Image image = LoganSquare.parse(json,  
    Image.class);
```

- Simple serializing methods

// Serialize it to an OutputStream

```
OutputStream os = //...
```

```
LoganSquare.serialize(image, os);
```

// Serialize it to a String

```
String json =
```

```
    LoganSquare.serialize(image);
```

- Small library
- Supports custom types
- **Compile-time**

Async management: Android Priority Job Queue (Job Manager)

Async management: Android Priority Job Queue (Job Manager)

- Job queue to easily schedule background tasks
- Inspired by a Google I/O 2010 talk on REST client applications
- Easy to declare a new tasks (extends Job) and configure it

```
public class PostTweetJob extends Job {  
    public static int PRIORITY = 1;  
  
    private String mText;  
  
    public PostTweetJob(String text) {  
        super(new Params(PRIORITY)  
            .requireNetwork()  
            .persist()));  
        mText = text;  
    }  
    //...  
}
```

```
public class PostTweetJob extends Job {  
    //...  
    @Override  
    public void onAdded() {  
        }  
  
    @Override  
    public void onRun() throws Throwable  
    {  
        webservice.postTweet(mText);  
    }  
    //...  
}
```

- Job manager configuration

```
Configuration configuration =
```

```
    new Configuration.Builder(poContext)
        .minConsumerCount(1)
        .maxConsumerCount(3)
        .loadFactor(3) // 3 jobs per
                       consumer
        .consumerKeepAlive(120)
        .build();
```

```
JobManager jobManager =
```

```
    new JobManager(context,
        configuration);
```

- Simple way to create and enqueue a task

```
PostTweetJob postTweetJob =  
    new PostTweetJob("test");  
  
jobManager  
    .addJobInBackground(postTweetJob);
```

Result propagation: EventBus

Result propagation: EventBus

- Based on the publisher/subscriber pattern (loose coupling)
- Communication between application components
- Small library
- Thread delivery
- Convenient Annotation based API

Set-up

- Create an event class

```
public class EventQueryDidFinish
```

- Register your subscriber...

```
eventBus.register(this);
```

- ...and unregister if needed:

```
eventBus.unregister(this);
```


- Declare subscribing method

```
@Subscribe(threadMode = ThreadMode.MAIN)
public void onEventQueryDidFinish(
    EventQueryDidFinish event) {
    //...
}
```

- Post event

```
EventQueryDidFinish event = //...
eventBus.post(event);
```

Conclusion

Conclusion

- Highly based on Java annotations
- Write less code
- Multiple ways to configure it
- Focused on performance and UX consistency