Retrofit is one of the most popular HTTP Client Library for Android as a result of its simplicity and its great performance compare to the others.

Anyway its weakness is there is no any straight way to cancel the ongoing transaction in Retrofit 1.x. If you want to do that you have to call it on Thread and kill it manually which is quite hard to manage.

Square gave a promise years ago that this feature will be available on Retrofit 2.0 but years passed, there is still no updated news on this.

Until last week, Retrofit 2.0 just passed its Release Candidate stage to Beta 1 and has been publicly launched to everyone. After giving it a try, I must say that I am quite impressed on its new pattern and its new features. There are a lot of changes in the good way. I will describe those in this article. Let's get started !

Same Old Package with New Version

If you want to import Retrofit 2.0 into your project, add this line to your `build.gradle` in `dependencies` section.

```
compile 'com.squareup.retrofit2:retrofit:2.0.0-beta4'
```

Sync your gradle files and you can now use Retrofit 2.0 =)

And as you see, Retrofit 2 package name is not the same as the previous version. It's now `com.squareup.retrofit2`.

New Service Declaration. No more Synchronous and Asynchronous.

In regard to service interface declaration in Retrofit 1.9, if you want to declare a synchronous function, you have to declare like this:

```
/* Synchronous in Retrofit 1.9 */

public interface APIService {

    @POST("/list")
    Repo loadRepo();

}
```

And you have to declare an asynchronous one like this:

```
/* Asynchronous in Retrofit 1.9 */

public interface APIService {

    @POST("/list")
    void loadRepo(Callback<Repo> cb);

}
```

But on **Retrofit 2.0**, it is far more simple since you can declare with only just a single pattern.

```
import retrofit.Call;

/* Retrofit 2.0 */

public interface APIService {

    @POST("/list")
    Call<Repo> loadRepo();

}
```

The way to call a created service is also changed into the same pattern as OkHttp. To call is as a synchronous request, just call `execute` or call `enqueue` to make an asynchronous request.

Synchronous Request

```
// Synchronous Call in Retrofit 2.0

Call<Repo> call = service.loadRepo();
Repo repo = call.execute();
```

The source code above will block the thread so you*cannot* call it on Main Thread in Android or you will face `NetworkOnMainThreadException`. If you want to call `execute` method, you have to do it on background thread.

Asynchronous Request

```
// Asynchronous Call in Retrofit 2.0

Call<Repo> call = service.loadRepo();
call.enqueue(new Callback<Repo>() {
    @Override
    public void onResponse(Response<Repo> response) {
        // Get result Repo from response.body()
    }

    @Override
    public void onFailure(Throwable t) {

    }
});
```

The above code will make a request in the background thread and retreive a result as an Object which you can extract from response with `response.body()` method. Please note that those call methods: `onResponse` and `onFailure` will be called in**Main Thread**.

I suggest you to use `enqueue`. It fits Android OS behavior best.

Ongoing Transaction Cancellation

The reason behind the service pattern changing to `Call` is to make the ongoing transaction be able to be cancelled. To do so, just simply call `call.cancel()`

```
call.cancel();
```

The transaction would be cancelled shortly after that. Easy, huh? =D

New Service Creation. Converter is now excluded from Retrofit.

In Retrofit 1.9, GsonConverter is included in the package and is automatically initiated upon `RestAdapter` creation. As a result, the json result from server would be automatically parsed to the defined Data Access Object (DAO).

But in Retrofit 2.0, Converter is ***not*** included in the package anymore. You need to plug a Converter in yourself or Retrofit will be able to accept only the String result. As a result, Retrofit 2.0 doesn't depend on Gson anymore.

If you want to accept json result and make it parse into DAO, you have to summon Gson Converter as a separate dependency.

```
compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
```

And plug it in through `addConverterFactory`. Please note that `RestAdapter` is now also renamed to `Retrofit`.

```
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://api.nuuneoi.com/base/")
                .addConverterFactory(GsonConverterFactory.create())
                .build();

        service = retrofit.create(APIService.class);
```

Here is the list of official Converter modules provided by Square. Choose one that fits your requirement best.

Gson: `com.squareup.retrofit:converter-gson`
Jackson: `com.squareup.retrofit:converter-jackson`
Moshi: `com.squareup.retrofit:converter-moshi`
Protobuf: `com.squareup.retrofit:converter-protobuf`
Wire: `com.squareup.retrofit:converter-wire`
Simple XML: `com.squareup.retrofit:converter-simplexml`

You also can create a custom converter yourself by implementing a Converter.Factory interface.

I support this new pattern. It makes Retrofit more clear what it actually does.

Custom Gson Object

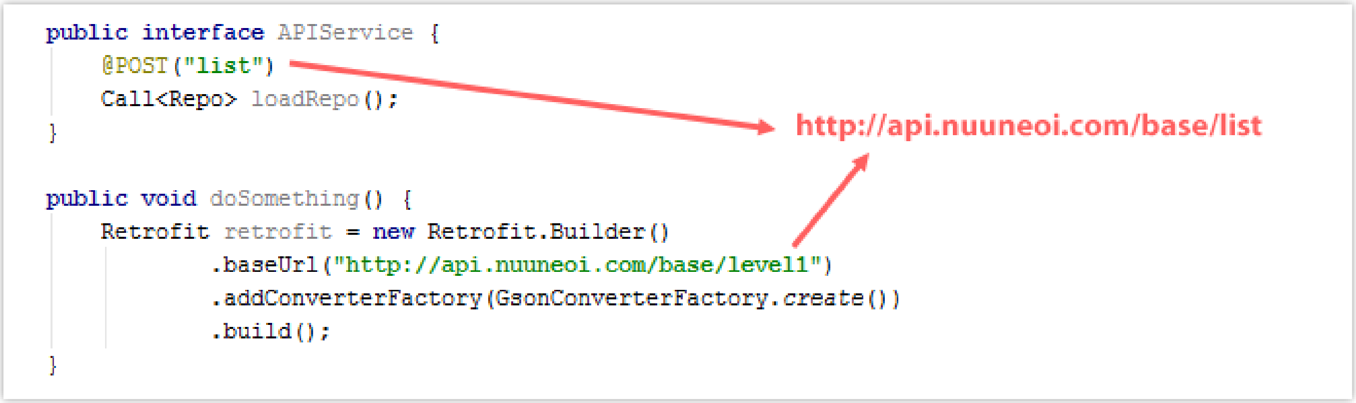
In case you need to adjust some format in json, for example, Date Format. You can do that by creating a Gson object and pass it to `GsonConverterFactory.create()`

```
        Gson gson = new GsonBuilder()
                .setDateFormat("yyyy-MM-dd'T'HH:mm:ssZ")
                .create();

        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://api.nuuneoi.com/base/")
                .addConverterFactory(GsonConverterFactory.create(gson))
                .build();

        service = retrofit.create(APIService.class);
```

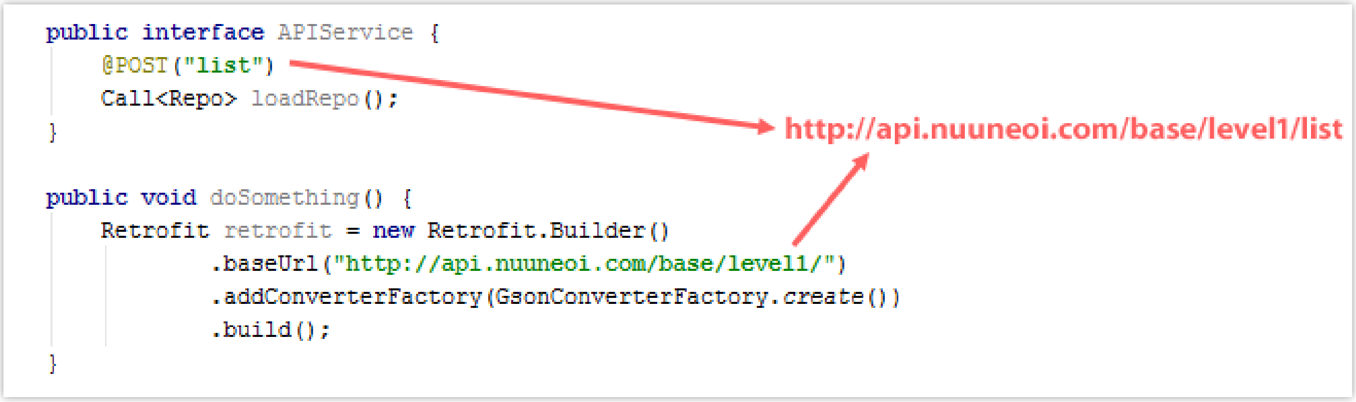
Done.

New URL resolving concept. The same way as <a href>

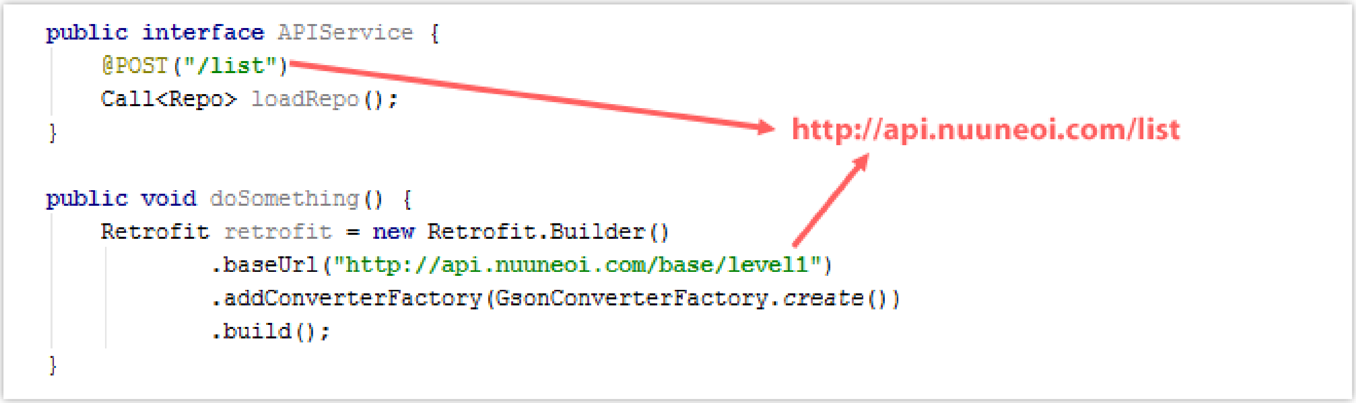
Retrofit 2.0 comes with new URL resolving concept. Base URL and @Url have not just simply been combined together but have been resolved the same way as what `<a href="...">` does instead. Please take a look for the examples below for the clarification.



```
public interface APIService {
    @POST("list")
    Call<Repo> loadRepo();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl("http://api.nuuneoi.com/base/level1")
            .addConverterFactory(GsonConverterFactory.create())
            .build();
}
```



```
public interface APIService {
    @POST("list")
    Call<Repo> loadRepo();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl("http://api.nuuneoi.com/base/level1/")
            .addConverterFactory(GsonConverterFactory.create())
            .build();
}
```



```
public interface APIService {
    @POST("/list")
    Call<Repo> loadRepo();
}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl("http://api.nuuneoi.com/base/level1")
            .addConverterFactory(GsonConverterFactory.create())
            .build();
}
```

Here is my suggestion on the new URL declaration pattern in Retrofit 2.0:

- **Base URL**: always ends with /

- **@Url**: DO NOT start with /

for instance

```
public interface APIService {

    @POST("user/list")
    Call<Users> loadUsers();

}

public void doSomething() {
    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl("http://api.nuuneoi.com/base/")
            .addConverterFactory(GsonConverterFactory.create())
            .build();

    APIService service = retrofit.create(APIService.class);
}
```

`loadUsers` from code above will fetch data from **`http://api.nuuneoi.com/base/user/list`**

Moreover we also can declare a full URL in `@Url` in Retrofit 2.0:

```
public interface APIService {

    @POST("http://api.nuuneoi.com/special/user/list")
    Call<Users> loadSpecialUsers();

}
```

Base URL will be ignored for this case.

You will see that there is a major change on URL resolving. It is totally different from the previous version. If you want to move your code to Retrofit 2.0, don't forget to fix those URLs part of code.

OkHttp is now required

OkHttp is set to optional in Retrofit 1.9. If you want to let Retrofit use OkHttp as HTTP connection interface, you have to manually include `okhttp` as a dependency yourself.

But in Retrofit 2.0, OkHttp is now required and is automatically set as a dependency. The code below is snapped from pom file of Retrofit 2.0. You have no need to do anything.

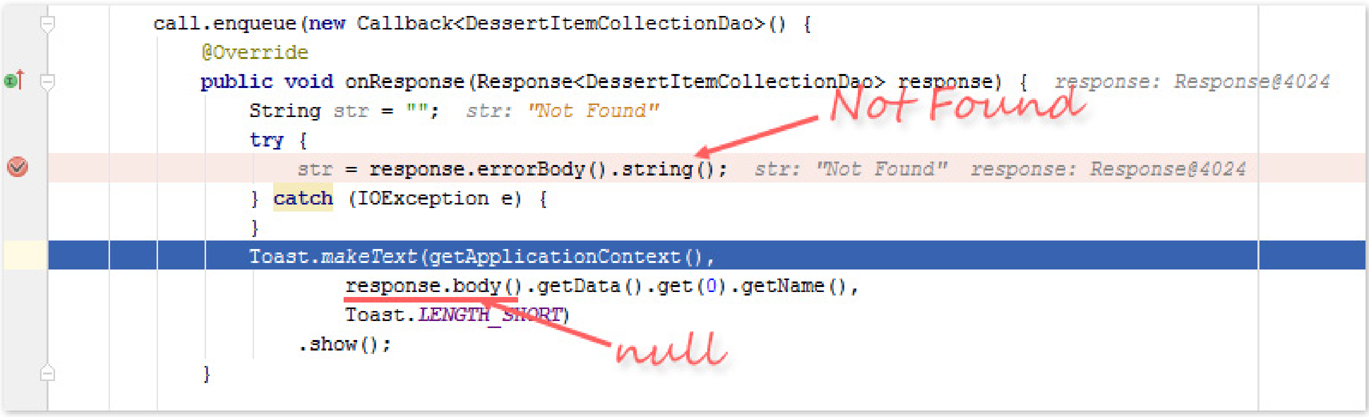
```
<dependencies>
  <dependency>
    <groupId>com.squareup.okhttp</groupId>
    <artifactId>okhttp</artifactId>
  </dependency>

  ...
</dependencies>
```

OkHttp is automatically used as a HTTP interface in Retrofit 2.0 in purpose to enabling the OkHttp's `Call` pattern as decribed above.

onResponse is still called eventhough there is a problem with the response

In Retrofit 1.9, if the fetched response couldn't be parsed into the defined Object, `failure` will be called. But in Retrofit 2.0, whether the response is be able to parse or not, `onResponse` will be always called. But in the case the result couldn't be parsed into the Object, `response.body()` will return as null. Don't forget to handle for the case.

If there is any problem on the response, for example, 404 Not Found. `onResponse` will also be called. You can retrieve the error body from `response.errorBody().string()`.



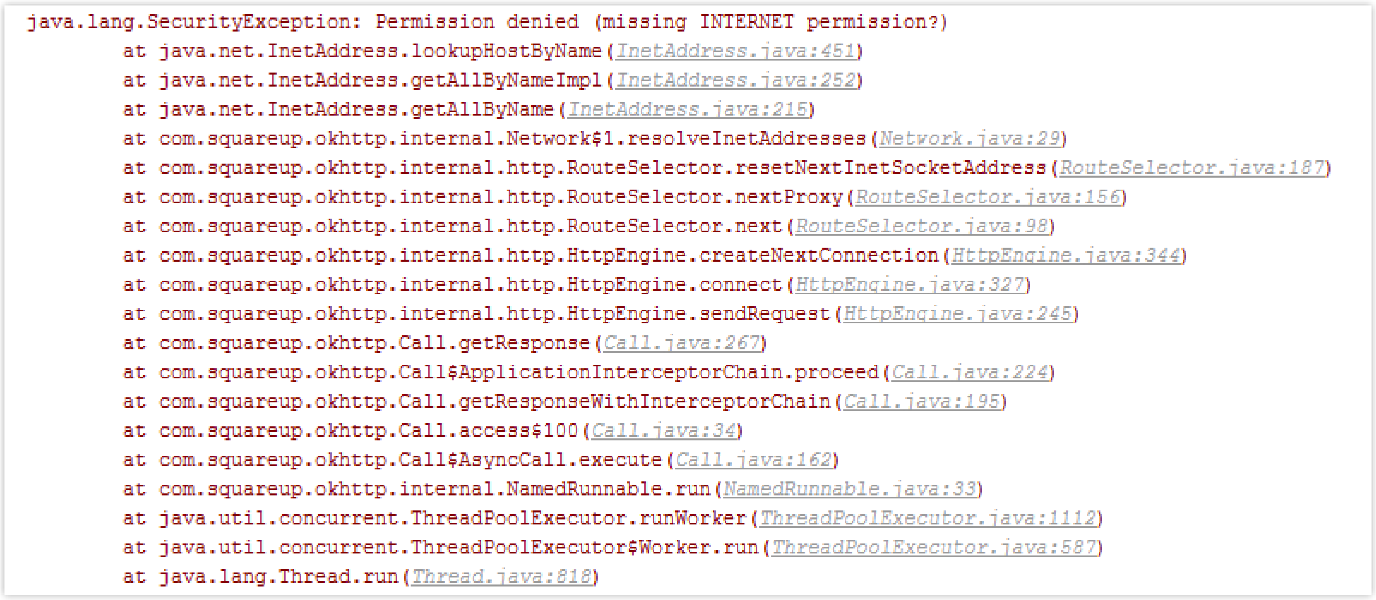
```
call.enqueue(new Callback<DessertItemCollectionDao>() {
    @Override
    public void onResponse(Response<DessertItemCollectionDao> response) {   response: Response@4024
        String str = "";   str: "Not Found"
        try {
            str = response.errorBody().string();   str: "Not Found"   response: Response@4024
        } catch (IOException e) {
        }
        Toast.makeText(getApplicationContext(),
                response.body().getData().get(0).getName(),
                Toast.LENGTH_SHORT)
            .show();
    }
```

Response/Failure logic is quite different from Retrofit 1.9. Be careful on handling for all the cases if you decide to move to Retrofit 2.0.

Missing INTERNET Permission cause SecurityException throwing

In Retrofit 1.9, if you forget to add INTERNET permission into your `AndroidManifest.xml` file. Asynchronous request will immediately fall into `failure` callback method with **PERMISSION DENIED** error message. None of exception is thrown.

But in Retrofit 2.0, when you call `call.enqueue` or `call.execute`, `SecurityException` will be immediately thrown and may cause crashing if you do not handle the case with try-catch.



```
java.lang.SecurityException: Permission denied (missing INTERNET permission?)
        at java.net.InetAddress.lookupHostByName(InetAddress.java:451)
        at java.net.InetAddress.getAllByNameImpl(InetAddress.java:252)
        at java.net.InetAddress.getAllByName(InetAddress.java:215)
        at com.squareup.okhttp.internal.Network$1.resolveInetAddresses(Network.java:29)
        at com.squareup.okhttp.internal.http.RouteSelector.resetNextInetSocketAddress(RouteSelector.java:187)
        at com.squareup.okhttp.internal.http.RouteSelector.nextProxy(RouteSelector.java:156)
        at com.squareup.okhttp.internal.http.RouteSelector.next(RouteSelector.java:98)
        at com.squareup.okhttp.internal.http.HttpEngine.createNextConnection(HttpEngine.java:344)
        at com.squareup.okhttp.internal.http.HttpEngine.connect(HttpEngine.java:327)
        at com.squareup.okhttp.internal.http.HttpEngine.sendRequest(HttpEngine.java:245)
        at com.squareup.okhttp.Call.getResponse(Call.java:267)
        at com.squareup.okhttp.Call$ApplicationInterceptorChain.proceed(Call.java:224)
        at com.squareup.okhttp.Call.getResponseWithInterceptorChain(Call.java:195)
        at com.squareup.okhttp.Call.access$100(Call.java:34)
        at com.squareup.okhttp.Call$AsyncCall.execute(Call.java:162)
        at com.squareup.okhttp.internal.NamedRunnable.run(NamedRunnable.java:33)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1112)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:587)
        at java.lang.Thread.run(Thread.java:818)
```

The behavior is just like the same when you manually call `HttpURLConnection`. Anyway this issue is not a big deal since when INTERNET permission is added into AndroidManifest.xml, there is nothing to concern anymore.

Use an Interceptor from OkHttp

On Retrofit 1.9 you could use `RequestInterceptor` to intercept a Request but it is already removed on Retrofit 2.0 since the HTTP connection layer has been moved to OkHttp.

As a result, we have to switch to an `Interceptor` from **OkHttp** from now on. First you have to create a `OkHttpClient` object with an Interceptor like this:

```
OkHttpClient client = new OkHttpClient();
client.interceptors().add(new Interceptor() {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Response response = chain.proceed(chain.request());

        // Do anything with response here

        return response;
    }
});
```

And the pass the created `client` into Retrofit's Builder chain.

```
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://api.nuuneoi.com/base/")
                .addConverterFactory(GsonConverterFactory.create())
                .client(client)
                .build();
```

That's all.

To learn more about what OkHttp Interceptor can do, please browse into OkHttp Interceptors.

Certificate Pinning

As same as an Interceptor, creation of an OkHttp client instance is required if you want to apply a Certificate Pinning with your connection. Here is the example code snippet. First, defines an OkHttp client instance with Certificate Pinning information:

```
OkHttpClient client = new OkHttpClient.Builder()
        .certificatePinner(new CertificatePinner.Builder()
                .add("publicobject.com", "sha1/DmxUShsZuNiqPQsX2Oi9uv2sCnw=")
                .add("publicobject.com", "sha1/SXxoaOSEzPC6BgGmxAt/EAcsajw=")
                .add("publicobject.com", "sha1/blhOM3W9V/bVQhsWAcLYwPU6n24=")
                .add("publicobject.com", "sha1/T5x9IXmcrQ7YuQxXnxoCmeeQ84c=")
                .build())
        .build();
```

Assign the OkHttp client created within Retrofit builder chain.

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://api.nuuneoi.com/base/")
        .addConverterFactory(GsonConverterFactory.create())
        .client(client)
        .build();
```

For more information about sha1 hash for Certificate Pinning ... Google would help a lot, just simple search for it how to achieve that piece of data.

RxJava Integration with CallAdapter

Beside declaring interface with `Call<T>` pattern, we also could declare our own type as well, for example, `MyCall<T>`. The mechanic is called "`CallAdapter`" which is available on Retrofit 2.0

There is some ready-to-use CallAdapter module available from Retrofit team. One of the most popular module might be CallAdapter for **RxJava**which will return as `Observable<T>`. To use it, two modules must be included as your project's dependencies.

```
    compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2'
    compile 'io.reactivex:rxandroid:1.0.1'
```

Sync Gradle and add `addCallAdapterFactory` in Retrofit Builder chain like this:

```
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("http://api.nuuneoi.com/base/")
                .addConverterFactory(GsonConverterFactory.create())
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
                .build();
```

Your Service interface is now able to return as `Observable<T>`!

```
public interface APIService {

    @POST("list")
    Call<DessertItemCollectionDao> loadDessertList();

    @POST("list")
    Observable<DessertItemCollectionDao> loadDessertListRx();

}
```

You can use it in the exact same RxJava way. In addition, if you want to let code inside subscribe part called on Main Thread, `observeOn(AndroidSchedulers.mainThread())` is needed to be added to the chain.

```
        Observable<DessertItemCollectionDao> observable = service.loadDessertListRx();

        observable.subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .unsubscribeOn(Schedulers.io())
            .subscribe(new Subscriber<DessertItemCollectionDao>() {
                @Override
                public void onCompleted() {
                    Toast.makeText(getApplicationContext(),
                            "Completed",
                            Toast.LENGTH_SHORT)
                        .show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getApplicationContext(),
                            e.getMessage(),
                            Toast.LENGTH_SHORT)
                        .show();
                }

                @Override
                public void onNext(DessertItemCollectionDao dessertItemCollectionDao) {
                    Toast.makeText(getApplicationContext(),
                            dessertItemCollectionDao.getData().get(0).getName(),
                            Toast.LENGTH_SHORT)
                        .show();
                }
            });
```

Done ! I believe that RxJava fan is very satisfying with this change =D

Conclusion

There are also some other changes, you can check for the official Change Log for more details. Anyway I believe that I have already covered the main issues in this article.

You may be curious that is it time to move to Retrofit 2.0 yet? Since it is still in the beta stage so you may want to stay with 1.9 first except you are an early adopter like me, Retrofit 2.0 works pretty great and there is no any bug found yet based on my own experiment.

Please note that Retrofit 1.9 official document was already removed from Square github website. I suggest you to start studying for Retrofit 2.0 right now and consider moving to the latest version in very near future. =D