Retrofit 2

What the IOException?!

- HttpException is more accurate

Retrofit 2

~~What the IOException?!~~

What the HttpException?!

- When planning IOException sounded cooler

Ash Davies

@DogmaticCoder

- Let me remind you of a quote from a couple of years back

"Retrofit 2 will be out by the end of this year"

— Jake Wharton (Droidcon NYC 2014)

- Cool!

"Retrofit 2 will be out by the end of this year"

— Jake Wharton (Droidcon NYC 2015)

- Erm, ok?

"Retrofit 2.0.0 is now released!"

— @JakeWharton (11 Mar 2016)

– Finally! Lets get going!


Finally!

- Over at ImmobilienScout24 GmbH we recently decided to update to Retrofit 2


BRILLIANT IDEA
MAYOR.
SOUTH PARK
COLORADO

Retrofit2: Dependencies

```
dependencies {

  // Retrofit 1.9
  compile 'com.squareup.retrofit:retrofit:1.9.0'

  // Retrofit 2.1
  compile 'com.squareup.retrofit2:retrofit:2.1.0'

}
```

- Retrofit 2 is distributed with a package name
- Meaning you include both libraries in your project

Method Counts

Retrofit2: Dependencies

Retrofit 1.9 (1997)
Retrofit (776)
Gson (1231)

Retrofit 2.1 (3349)
Retrofit (508)
OkHttp (2180)
Okio (661)

- Considerable difference because of dependency on OkHttp

OkHttp

- Retrofit inherits OkHttp call pattern
- Uses request elements from OkHttp too
- Interceptor, Response, RequestBody

Awesome!

What's Next?

- Lets build our rest adapter

Retrofit2: RestAdapter

```
// Retrofit 1.9
RestAdapter adapter = new RestAdapter.Builder()
   .setClient(new Ok3Client(okHttpClient))
   .setEndpoint("...")
   .build();

// Retrofit 2.1
Retrofit retrofit = new Retrofit.Builder()
   .client(okHttpClient)
   .baseUrl("...")
   .build();
```

- `RestAdapter` renamed to `Retrofit`
- No Retrofit client wrapper (Ok3Client)
- `setEndpoint` renamed to `baseUrl`

Converters

- Converters deal with the serialisation of your data objects
- Factories no longer dependent on Gson package
- Are parameterised and generated from a factory

Retrofit2: Converters

```
compile 'com.squareup.retrofit2:converter-gson:2.1.0'
```

- Retrofit 1.9 included GsonConverter and had gson as a dependency
- Retrofit 2.1 does not include any converters automatically

Retrofit2: Converters

```
compile 'com.squareup.retrofit2:converter-gson:2.1.0'

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

- This covers both your request and response body converters

Retrofit2: Converters

```
compile 'com.squareup.retrofit2:converter-gson:2.1.0'

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

moshi, scalars, simplexml, wire, jackson, protobuf

- Other serialisation converters can be used
- No duplication of converters using an alternative

Multiple Converters

- Using multiple converters is similar to Retrofit1
- Converter Factory is not dependent on Gson either

Retrofit2: Multiple Converters

→ Checks every converter sequentially

- Retrofit2 checks every converter that is capable of dealing with the data type
- If it can't understand the data Retrofit moves to the next one

Retrofit2: Multiple Converters

→ Checks every converter sequentially

→ Register converter factories in order

- Retrofit will take first successful converter
- Register your special converter factories first
- General converters like Gson last

Retrofit2: Multiple Converters

→ Checks every converter sequentially

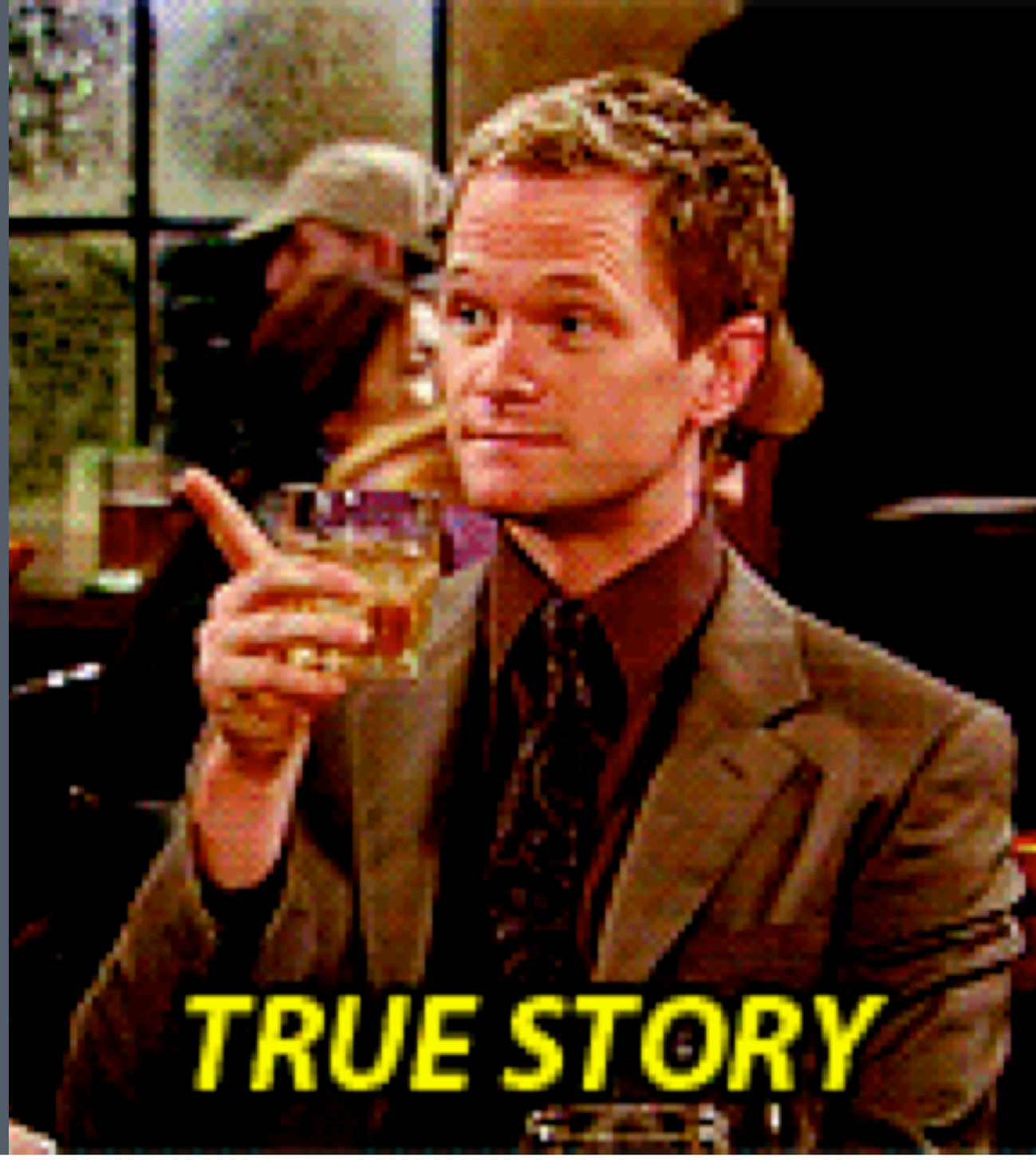
→ Register converter factories in order

→ Create custom `retrofit2.Converter.Factory`

- You can create custom converter factories
- For use with custom serialisation methods
- If you have legacy code or manual json converters
- Any questions on this directed as @alosdev

"I love Factories!"

— @alosdev



- Original request logging is no longer included with Retrofit

Retrofit2: Logging

```
dependencies {
   compile 'com.squareup.okhttp3:logging-interceptor:3.4.1'
}
```

- Since Retrofit now uses OkHttp as its client

Retrofit2: Logging

```
HttpLoggingInterceptor logger = new HttpLoggingInterceptor();
logger.setLevel(HttpLoggingInterceptor.Level.BODY);

OkHttpClient client = new OkHttpClient.Builder()
        .addInterceptor(logger)
        .build();
```

- Logging must be performed via an interceptor

Adapters

Adapters

Call -> Asynchronous Consumer

- Adapters convert a standard call to another asynchronous consumer
- Unlike Retrofit1 call adapters do not come included

Retrofit2: Adapters

```
// RxJava: Observable<T>
com.squareup.retrofit2:adapter-rxjava
```

- RxJava to use with observables

Retrofit2: Adapters

```
// RxJava: Observable<T>
com.squareup.retrofit2:adapter-rxjava

// RxJava2: Flowable<T>
com.jakewharton.retrofit:retrofit2-rxjava2-adapter
```

- RxJava2 provided by Jake Wharton until RxJava2 stable release

Retrofit2: Adapters

```
// RxJava: Observable<T>
com.squareup.retrofit2:adapter-rxjava

// RxJava2: Flowable<T>
com.jakewharton.retrofit:retrofit2-rxjava2-adapter

// Java8: CompletableFuture<T>
com.squareup.retrofit:converter-java8
```

- Java8 futures for Java projects (not available on Jack compiler)

Retrofit2: Adapters

```
// RxJava: Observable<T>
com.squareup.retrofit2:adapter-rxjava

// RxJava2: Flowable<T>
com.jakewharton.retrofit:retrofit2-rxjava2-adapter

// Java8: CompletableFuture<T>
com.squareup.retrofit:converter-java8

// Guava: ListenableFuture<T>
com.squareup.retrofit:converter-guava
```

- Guava futures if you MultiDex and hate your users

- So good so far...
- What else?

Services

- There is no difference between asynchronous and synchronous call definitions

Retrofit2: Services

```
/* Retrofit 1.9 */
public interface Service {

  // Synchronous
  @GET("/articles")
  List<Article> articles();

  // Asynchronous
  @GET("/articles")
  void articles(Callback<List<Article>> callback);
}
```

- Retrofit 1 allows you to define both in the service definitions

Retrofit2: Services

```
/* Retrofit 2.1 */
public interface Service {

  @GET("articles")
  Call<List<Article>> articles();
}
```

- Retrofit 2 does not allow this in the interface definition
- Responses are now encapsulated in a parameterized call
- Take another look at the url in this service

Retrofit2: Services

```
/* Retrofit 1.9 */
@GET("/articles")

/* Retrofit 2.1 */
@GET("articles")
```

- Base url resolution provided by HttpUrl.resolve()
- Use relative urls for your partial endpoints

Retrofit2: Services

```
public interface AwsService {

   @GET
   public Call<File> getImage(@Url String url);
}
```

- Retrofit now supports dynamic urls

Retrofit2: Services

```
public interface AwsService {

  @GET
  public Call<File> getImage(@Url String url);
}
```

Useful when working with external Services

- If you need to upload an image to AWS
- Needed to create a new retrofit client

Retrofit2: Services

```
/* Retrofit 2.1 */
Call<List<Article>> call = service.articles();

// Synchronous
call.execute();

// Asynchronous
call.enqueue();
```

- Call can be made synchronously with `execute()` or asynchronously with `enqueue()`

Retrofit2: Cancel Requests

```
/* Retrofit 2.1 */

Call<ResponseBody> call = service.get("...");

call.enqueue(new Callback<ResponseBody>() { ... });

call.cancel();
```

- Either because the user has navigated somewhere where the request isn't necessary
- Cancelling on Retrofit 2 is as simple as calling a method on the call
- and you can check if the call was cancelled with `isCancelled()`


Citytv

Rx all the things!

- For the rest of this I'm going to assume the use of RxJava adapters

- Life is good right now
- This migration seems easy!

- How would retrofit1 handle server errors

Retrofit2: Errors

- Lets say we have a standard user journey
- But the user puts in a wrong email address

Retrofit2: Errors

```
{
    statusCode: 400,
    message: "Malformed email"
}
```

- Here the server gives us some useful information
- This may also localised using your accept headers

```
/* Retrofit 1.9 */
service.login(username, password)
  .subscribe(user -> {
    session.storeUser(user);
    view.gotoProfile();
  }, throwable -> {
    if (throwable instanceof RetrofitError) {
      processServerError(
        ((RetrofitError) throwable).getBodyAs(ServerError.class)
      );
    }

    view.onError(throwable.getMessage());
});
```

- How would you consume this error in Retrofit 1?
- Goto user profile on success, process server error on failure

```
FAILURE: Build failed with an exception.
`error: cannot find symbol class RetrofitError`
```

- What happened to RetrofitError?
- Has it moved? renamed?
- What can I do about this?

* goes to stackoverflow.com *

RetrofitError is dead.

Long live HttpException.

RetrofitError

- Lets first take a look at RetrofitError
- What was it doing for us and why it gets such a bad rep

Retrofit: RetrofitError

```
/* Retrofit 1.9: RetrofitError */
public Object getBodyAs(Type type) {
  if (response == null) {
    return null;
  }
  TypedInput body = response.getBody();
  if (body == null) {
    return null;
  }
  try {
    return converter.fromBody(body, type);
  } catch (ConversionException e) {
    throw new RuntimeException(e);
  }
}
```

- RetrofitError allows you to cast the error body response
- Violation of single responsibility principle

"How do you know if it was a conversion error, network error, random error inside Retrofit? Yuck."

— Jake Wharton (Jul 25, 2013)

Retrofit: RetrofitError

```
/** Identifies the event kind which triggered a {@link RetrofitError}. */
public enum Kind {
  /** An {@link IOException} occurred while communicating to the server. */
  NETWORK,
  /** An exception was thrown while (de)serializing a body. */
  CONVERSION,
  /** A non-200 HTTP status code was received from the server. */
  HTTP,
  /**
   * An internal error occurred while attempting to execute a request. It is best practice to
   * re-throw this exception so your application crashes.
   */
  UNEXPECTED
}
```

Introduced in Retrofit v1.7 (Oct 8, 2014)

"I find that API to awful (which I'm allowed to say as the author)"

— Jake Wharton (Nov 7, 2015)

- Forces creating messy bug prone code in the caller

What Now?

- So what do we do now?
- Lets revisit our login code

```
/* Retrofit 2.1 */
service.login(username, password)
  .subscribe(user -> {
    session.storeUser(user);
    view.gotoProfile();
  }, throwable -> {
    // Now What?!
  });
```

- How can we react to this throwable?

Exceptions

- In Retrofit2 there are two main exception

Retrofit2: Exceptions

IOException: Network errors

- Loss of connectivity
- Serialisation errors
- Hard to know which

Retrofit2: Exceptions

HttpException: Non 2xx responses

- Server errors giving a non-2xx responses

HttpException

Retrofit2: HttpException

→ Included in Retrofit2 adapters

Retrofit2: HttpException

→ Included in Retrofit2 adapters

→ Contains a response object

Retrofit2: HttpException

→ Included in Retrofit2 adapters

→ Contains a response object

→ Response is not serialisable

IOException

Retrofit2: IOException

RetrofitError.Kind.NETWORK

RetrofitError.Kind.CONVERSION

- Detecting serialisation issues is pretty hard
- IOExceptions are thrown for both network errors and conversions

Retrofit: IOExceptions

isConnectedOrConnecting();

- You can typically detect connection issues with the ConnectivityManager
- Be careful not to pollute your error handling code however
- So what will our throwable look like?

```
throwable -> {
  if (throwable instanceof HttpException) {
    // non-2xx error
  }

  else if (throwable instanceof IOException) {
    // Network or conversion error
  }

  else {
    // ¯\_(ツ)_/¯
  }
}
```

Cool story bro...

- Ok cool so we know our exceptions
- How can we deal with it?

```java
@AutoValue
public abstract class ServerError extends RuntimeException {

  public abstract int getStatusCode();
  public abstract String message();
}
```

- Lets say we have a server error object
- For brevity here I'm using AutoValue

```
public class ServerErrorProcessor {

  private final Converter<ResponseBody, ServerError> converter;

  public ServerErrorProcessor(Retrofit retrofit) {
    this(retrofit.responseBodyConverter(ServerError.class, new Annotation[0]));
  }

  private ServerErrorProcessor(Converter<ResponseBody, ServerError> converter) {
    this.converter = converter;
  }

  public <T> Function<Throwable, Publisher<? extends T>> convert() {
    return throwable -> {
      if (throwable instanceof HttpException) {
        Response response = ((HttpException) throwable).response();
        return Flowable.error(converter.convert(response.errorBody()));
      }

      return Flowable.error(throwable);
    };
  }
}
```

- Creates a response body converter from the retrofit instanceof
- Provides a Flowable function to convert HttpException to a server error

```
public static class UserClient {

  private final ErrorProcessor processor;
  private final UserService service;

  public UserClient(Retrofit retrofit) {
    this(retrofit.create(UserService.class), new ErrorProcessor(retrofit));
  }

  private UserClient(UserService service, ErrorProcessor processor) {
    this.service = service;
    this.processor = processor;
  }

  public Flowable<User> login(String username, String password) {
    return service.login(username, password)
        .onErrorResumeNext(processor.convert());
  }
}
```

- Use Rx onErrorResumeNext to check for an HttpException
- HttpException has been converted to a parsed ServerError
- If you want to wrap the entire exception into a custom one...

"RetrofitException"

gist.github.com/koesie10/bc6c62520401cc7c858f

- Introduces Retrofit1 error handling in Retrofit2
- Doesn't really solve the problem that RetrofitError had

```
public static RetrofitException from(Throwable throwable) {
  if (throwable instanceof HttpException) {
    Response response = (HttpException) throwable).response();
    Request request = response.raw().request();

    return RetrofitException.http(request.url().toString(), response, retrofit);
  }

  if (throwable instanceof IOException) {
    return RetrofitException.network((IOException) throwable);
  }

  return RetrofitException.unexpected(throwable);
}
```

- Creates RetrofitException as clone of RetrofitError
- Can be created from HttpException or IOException

```
public class RxErrorHandlingCallAdapterFactory extends CallAdapter.Factory {
  private final RxJavaCallAdapterFactory original = RxJavaCallAdapterFactory.create();

  @Override
  public CallAdapter<?> get(Type returnType, Annotation[] annotations, Retrofit retrofit) {
    return new RxCallAdapterWrapper(retrofit, original.get(returnType, annotations, retrofit));
  }

  private static class RxCallAdapterWrapper implements CallAdapter<Observable<?>> {
    private final Retrofit retrofit;
    private final CallAdapter<?> wrapped;

    public RxCallAdapterWrapper(Retrofit retrofit, CallAdapter<?> wrapped) {
      this.retrofit = retrofit;
      this.wrapped = wrapped;
    }

    @Override
    public Type responseType() {
      return wrapped.responseType();
    }

    @Override
    public <R> Observable<?> adapt(Call<R> call) {
      return ((Observable) wrapped.adapt(call)).onErrorResumeNext(throwable -> {
        return Observable.error(RetrofitException.from(throwable));
      });
    }
  }
}
```

RxJava Adapters

- You can also take more control over the result
- RxJava adapters for different service responses
- Important to know which exceptions will be passed

RxJava Adapters

```
Observable<T> call();
```

- onNext with deserialised body for 2xx responses
- onError with HttpException for non-2xx responses
- onError with IOException for network calls

RxJava Adapters

```
Observable<Response<T>> call();
```

- onNext with a response object for all Http responses
- onError with IOException for network errors

RxJava Adapters

```
Observable<Result<T>> call();
```

- onNext with a result object for all Http responses and errors

Thank You