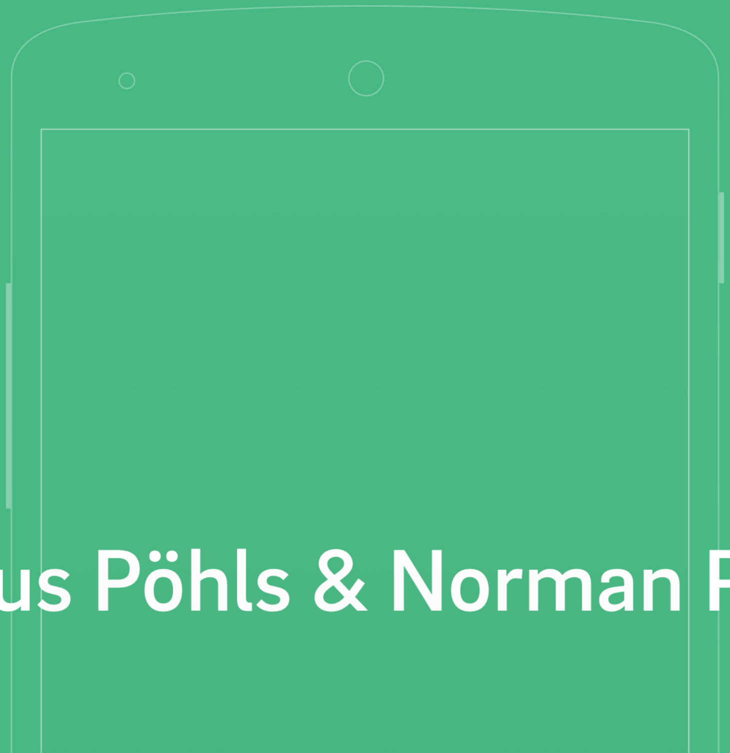
# Retrofit

Love Working With APIs
On Android



Marcus Pöhls & Norman Peitek

# Retrofit: Love Working with APIs on Android

Take delight in building API clients on Android.

Marcus Pöhls

This book is for sale at http://leanpub.com/retrofit-love-working-with-apis-on-android

This version was published on 2016-03-19

# Also By Marcus Pöhls

# Contents

**Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **i**
What Topics Are Waiting for You? . . . . . . . . . . . . . . . . . . . . . . . . . . . . i
Who Is This Book For? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ii
The Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii
Retrofit Book for Version 1.9 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

**Chapter 1 — Quick Start & Create a Sustainable REST Client** . . . . . . . . . . . . . . . . . **1**
Retrofit Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
Why Use Retrofit? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
Retrofit vs. OkHttp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
Quick Start: Add Retrofit to Your Project . . . . . . . . . . . . . . . . . . . . . . 2
Sustainable Android REST Client . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
Retrofit in Use . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**Chapter 2 — Requests** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **10**
API Declaration With Retrofit . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
Url Handling, Resolution and Parsing . . . . . . . . . . . . . . . . . . . . . . . . 11
Query Parameter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
Path Parameter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
Synchronous and Asynchronous Requests . . . . . . . . . . . . . . . . . . . . . . . 26
Send Objects in Request Body . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
Constant, Default and Logic Values for POST and PUT Requests . . . . . . . . . . . . 31
Send Data Form-Urlencoded . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Add Request Header . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
Reuse, Analyze & Cancel Requests . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**Chapter 3 — Data Mapping with Retrofit Converters** . . . . . . . . . . . . . . . . . . . **49**
Integrate a JSON Converter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
Integrate an XML Converter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Integrate Other Standard Formats . . . . . . . . . . . . . . . . . . . . . . . . . . 54
Dealing with Multiple Converters . . . . . . . . . . . . . . . . . . . . . . . . . . 55
Integrate Your Own Custom Converter . . . . . . . . . . . . . . . . . . . . . . . . 56

**Chapter 4 — Authentication** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **69**

Authentication Interceptor . . . . . . . . . . 69
Basic Authentication . . . . . . . . . . 70
Token Authentication . . . . . . . . . . 74
OAuth 2 . . . . . . . . . . 77
OAuth — Refresh Your Access Token . . . . . . . . . . 86
Hawk Authentication . . . . . . . . . . 91

**Chapter 5 — File Upload & Download . . . . . . . . . . 97**
File Upload . . . . . . . . . . 97
Upload of Multiple Files . . . . . . . . . . 101
Passing Multiple Parts Along a File with @PartMap . . . . . . . . . . 103
File Download . . . . . . . . . . 106

**Chapter 6 — Pagination on Android . . . . . . . . . . 112**
Pagination Response Status . . . . . . . . . . 112
Pagination Using Query Parameter . . . . . . . . . . 112
Pagination Using Link Header and Dynamic Urls (Like GitHub) . . . . . . . . . . 117
Pagination Using Range Header Fields (Like Heroku) . . . . . . . . . . 122

**Chapter 7 — Error Handling . . . . . . . . . . 129**
Error Handling Preparations . . . . . . . . . . 129
Simple Error Handler . . . . . . . . . . 130

**Chapter 8 — Logging . . . . . . . . . . 134**
Log Requests and Responses with Retrofit . . . . . . . . . . 134
Enabling Logging for Debug Builds Only . . . . . . . . . . 137

**Chapter 9 — Multiple Server Environments . . . . . . . . . . 140**
Product-Flavor-Dependent Base URLs . . . . . . . . . . 140
Change API Base URL at Runtime . . . . . . . . . . 143

**Chapter 10 — Client-Side Mocking of Server Responses . . . . . . . . . . 148**
Why Mocking Responses Make Sense . . . . . . . . . . 148
Create Your Own MockRetrofit Implementation . . . . . . . . . . 148
Customizing Network Behavior of Mocked Server Responses . . . . . . . . . . 153

**Chapter 11 — App Release Preparation . . . . . . . . . . 157**
Enable ProGuard . . . . . . . . . . 157
Configure ProGuard Rules for Retrofit . . . . . . . . . . 158
Obfuscated Stack Traces . . . . . . . . . . 158

**Outro . . . . . . . . . . 160**

**About the Book . . . . . . . . . . 161**

**Appendix — Retrofit 2 Upgrade Guide from 1.x . . . . . . . . . . . . . . . . . . . . . . . . . . 162**
Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
Maven & Gradle Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
RestAdapter —> Retrofit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
setEndpoint —> baseUrl . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
Base Url Handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
Dynamic Urls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
OkHttp Required . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
Interceptors Powered by OkHttp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167
Synchronous & Asynchronous Requests . . . . . . . . . . . . . . . . . . . . . . . . . . 168
Cancel Requests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
No Default Converter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
RxJava Integration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
No Logging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
Future Update: WebSockets in Retrofit 2.1 . . . . . . . . . . . . . . . . . . . . . . . 173
Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174

# Introduction

Due to the popularity of the Retrofit blog post series[1] published in the Future Studio blog, we've decided write a book on Retrofit. We're delighted about the amazing popularity of the Retrofit series! **Thanks a lot for all of the positive feedback, comments and encouragement!** Your kind words go a long way and keep us motivated to publish more content on Retrofit.

This is the second edition of our Retrofit book. We've updated the content including code examples to address **Retrofit 2**. The first edition was completely geared towards Retrofit 1. If you're using Retrofit 1 and want to read the first edition of this book, please visit this book's extras on Leanpub to download the package with `PDF`, `mobi` and `epub` files.

Within this book, we keep the kind of techy style from the tutorials to make this book a great resource for every developer working with Retrofit.

## What Topics Are Waiting for You?

You probably scanned the table of contents and know what to expect. Let me describe the chapters in short before we move on and dig deeper into Retrofit and its functionality.

**Covered Topics**:

- Introduction to Retrofit
- **Quick Start Guide** to jump right into Retrofit
- Create a **Sustainable Android REST Client**
- Extensive manipulation and customization of **requests**
- Comprehensive overview of **response converters and data mapping**
- Handling **Authentication** on Android (Basic, Token, OAuth, Hawk)
- Advanced File Handling, like **File Up- and Download**
- How to **handle errors** in your Android app
- Debug requests and response using **logging**
- App release preparation including ProGuard configuration

#### Covered Topics: Give Me the Deatails

The book starts out with an overview about what Retrofit is and how to prepare your Android project to use Retrofit. Further, we'll walk you through the setup on how to create a sustainable REST client basis. Additionally, we'll dive into the basics on how responses get mapped to Java objects and create

[1]https://futurestud.io/blog/retrofit-getting-started-and-android-client

the first client to perform a request against the GitHub API (learning by doing is an effective method :)).

Once we managed the jumpstart, we show you the details about Retrofit's requests: how to perform them synchronous and asynchronous and how to manipulate requests to your personal needs, like adding request parameters, path parameters, request payload and a lot more!

We'll also walk you through Retrofit responses, show you how to change the response converter and how to mock an API on Android itself.

Knowing the basics about Retrofit, we touch a common use case: authentication. Besides basic and token authentication, we'll explain how to use Retrofit for OAuth (including OAuth 2) and how to use OAuth's refresh token to get back a valid access token.

File handling can be kind of tricky, so let's take the road together! We guide you through file up- and downloads with Retrofit and show the actions required to send and receive different types of files like images and compressed packages (e.g. `ZIP`).

Last but not least: once you get your app (using Retrofit) out the door, you need to prepare the release for Google Play. This chapter digs into the correct configuration of ProGuard for Android projects integrating Retrofit and presents examplary rules to keep your app working correctly after distributing via Google Play.

## Who Is This Book For?

In short: **this book is for you**. We believe that every developer reading this book will take away new ideas on how to optimize their Android apps in the sense of interacting with API's or webservices using Retrofit. You'll recognize goodies that are applicable for the projects you're working on.

This book is for Android developers who want to receive a substantial overview and reference book on Retrofit. You'll benefit from clearly recognizable code examples related to your daily work with Retrofit.

### Rookie

If you're just starting out with Retrofit (or coming from another HTTP library like Android Asynchronous Http Client or even Volley) this book will show you all important parts on how to create sustainable REST clients on Android. The provided code snippets let you jumpstart and create your first successful API client within minutes!

### Expert

If you've already worked with Retrofit, you'll profit from our extensive code snippets and apply the learnings to your existing code base. Additionally, the book illustrates various use cases for different functionalities and setups like authentication against different backends, request composition, file up- and downloads, etc.!

## The Source Code

With the purchase of this book, you benefit from the sample project that await your download within the **extras** section on Leanpub. The sample project is an Android project based on gradle. You can directly touch and use classes that are only illustrated in excerpts within this book.

Check your Leanpub Library[2] and select **Retrofit: Love working with APIs on Android** to download the sample code base.

## Retrofit Book for Version 1.9

This is the second edition of the Retrofit book. It's fully focussed on Retrofit 2, no hint or code snippet that points to Retrofit 1. If you're interested in the first version of the book that based on Retrofit 1, please visit this book's **extras** page on Leanpub. As a reader of this book, the first version is — of course — also available for you!

Now, let's jump right in and get started with Retrofit!

[2]https://leanpub.com/user_dashboard/library

# Chapter 1 — Quick Start & Create a Sustainable REST Client

Within this chapter we're going through the basics of Retrofit, give a brief overview to jump-start with Retrofit in your project and create a REST client foundation that we'll enhance and apply within multiple chapters of this book.

Precisely, we start with a short overview of the Retrofit project and why you should make use of it. Afterwards, we show you how to prepare your Android project to utilize Retrofit. To get you hooked on the Retrofit train, we create a sustainable Android REST client including the assignment of a JSON response converter.

To clarify our usage of the term **Retrofit** throughout this book: we always refer to Retrofit 2. The content completely gears toward Retrofit 2 and we just keep it short by using Retrofit.

Remember, al ready mentioned in the Introduction chapter: if you're interested in the previous book version on Retrofit 1.9, we've added the book's PDF, mobi and ePub files within the extras of this book on Leanpub[3].

## Retrofit Overview

Retrofit is a »type-safe REST client for Android and Java«[4].

Retrofit abstracts your REST API into Java interfaces. You'll use annotations to describe your individual API endpoints and their HTTP requests. Support for URL parameter replacement (like query and path parameter) is integrated by default, as well as functionality for form-urlencoded and multipart requests. You can of course execute requests using the HTTP methods `GET`, `POST`, `PUT`, `PATCH`, `DELETE`. Anyway, this is literally just the tip of the iceberg and there's a lot more to discover behind Retrofit's scenes.

## Why Use Retrofit?

Working with APIs can cause a lot headaches due to various scenarios and edge cases. Problems already arise when just giving the low-level networking some thoughts. Handling connections and interruptions, caching, request retries and SSL handshakes. Also, the worry about worker threads

[3]https://leanpub.com/user_dashboard/library
[4]http://square.github.io/retrofit/

or runnables or even lovely AsyncTasks definitely causes a lot pain in the a**! Think about the pain and effort you need to put into your own implementation.

You'll safe yourself a lot of time and anger when leveraging a well thought-out and tested library like Retrofit. If you scrolled through the outline of this book, you've already an impression about the functionality and flexibility that comes with Retrofit. Benefit from the library and put your work to pieces where your attention is actually needed!

## Retrofit vs. OkHttp

Downright this book, we'll make use of OkHttp and show you how to achieve a solution leveraging the functionality of OkHttp. And the reason is simple: OkHttp is a pure HTTP/SPDY client responsable for any low-level network operation, caching, request and response manipulation, and many more. In contrast, Retrofit is a high-level REST abstraction build on top of OkHttp. Retrofit 2 is strongly coupled with OkHttp and makes intensive use of it. That's the reason why we sometimes need to borrow OkHttp's classes to accomplish the solution.

## Quick Start: Add Retrofit to Your Project

Now let's get our hands dirty and back to the keyboard. If you already created your Android project, just go ahead and start from the next paragraph („Internet Permission in Android's Manifest"). If not, create a new Android project in your favorite IDE. We use Android Studio[5] and prefer the Gradle build system, but you surely can use Maven as well.

### Internet Permission in Android's Manifest

We use Retrofit to perform HTTP requests against an API running on a server somewhere in the Internet. Executing those requests from an Android application requires the `Internet` permission to open network sockets. You need to define the permission within the `AndroidManifest.xml` file. If you didn't set the Internet permission yet, please add the following line within your manifest definition:

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

A common practice is, to add your app permissions as the first elements in your manifest file. The following snippet is an excerpt from the sample project you can find within the extras of this book on Leanpub.

[5]https://developer.android.com/sdk/index.html

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="android.retrofitbook.futurestud.io.fsretrofitbook"
          xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.INTERNET"/>

    …
```

## Define Dependency: Gradle or Maven

Now let's define Retrofit as a dependency for your project. Depending on your used build system, define Refrofit in your `build.gradle` or `pom.xml` file. When running the command to build your code, the build system will download and provide the library for your project.

At the time of this book being written, the latest Retrofit version is `2.0.0`. There has been a lot activity in the Retrofit repository on GitHub during the last weeks to finish the second major release. We'll keep this book update to future versions of Retrofit.

Ok, now let's add Retrofit as a dependency to our project:

**build.gradle**

```
dependencies {
    // Retrofit & OkHttp
    compile 'com.squareup.retrofit2:retrofit:2.0.0'
}
```

**pom.xml**

```xml
<dependency>
    <groupId>com.squareup.retrofit2</groupId>
    <artifactId>retrofit</artifactId>
    <version>2.0.0</version>
</dependency>
```

Sync your Gradle or Maven project to import the required packages for Retrofit. Retrofit 2 doesn't ship with an integrated response converter anymore. We need to manually add the desired response converters as a dependency to our project. The following section will show you how to add Gson for JSON response mapping to your project.

## JSON Response Mapping

As already mentioned, by default Retrofit 2 doesn't come with any response converter integrated. To add Google's Gson for JSON to Java object mapping, we need to add another Gradle or Maven dependency to our project. Update your `build.gradle` or `pom.xml` file appropriately to import Retrofit's Gson converter besides Retrofit.

**build.gradle**

```
dependencies {
    // Retrofit & OkHttp
    compile 'com.squareup.retrofit2:retrofit:2.0.0'
    compile 'com.squareup.retrofit2:converter-gson:2.0.0'
}
```

**pom.xml**

```
<dependency>
    <groupId>com.squareup.retrofit2</groupId>
    <artifactId>retrofit</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>com.squareup.retrofit2</groupId>
    <artifactId>converter-gson</artifactId>
    <version>2.0.0</version>
</dependency>
```

Once again, synchronize your project to import the Gson converter for Retrofit. In the following, you'll learn how to define a service interface to map a given API endpoint.

## Declare an API Interface

This part of the quick start guide is intended to get you kickstarted and how applicable Retrofit is for your Android app. Let's directly jump in and use a code example to make things approachable:

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

The snippet above defines a Java interface called `GitHubService` having only one method `reposForUser(…)`. The method and its parameters have Retrofit annotations which describe the behavior of this method.

The `@GET()` annotation explicitly defines that a GET request will be executed once the method gets called. Further, the `@GET()` definition takes a string parameter representing the endpoint url of your API. Additionally, the endpoint url can be defined with placeholders which get substituted by path parameters.

The method signature contains a `@Path()` annotation for the `user` parameter. This `@Path` annotation maps the provided parameter value during the method call to the path within the request url. The declared `{user}` part within the endpoint url will be replaced by the provided value of `username`. We'll catch up path parameters in more detail in a later chapter.

The snippet above only describes how to define your API on the client side. That's fine for now. We'll have a look at the practical example in a second. There, we're going to execute the actual API request.

## Sustainable Android REST Client

During the research for already existing Retrofit clients, the example repository of Bart Kiers[6] came up. Actually, it's an example for OAuth authentication with Retrofit. However, it provides all necessary fundamentals for a sustainable Android client. That's why we'll use it as a fundamental part and extend it during future chapters within this book (with functionality like authentication, logging, and many more).

The following class defines the basis of our Android client: **`ServiceGenerator`**.

### Service Generator

The **`ServiceGenerator`** is our API client's heart. In its current state, it only defines one method to create a basic REST client for a given class/interface which returns a service class from the interface. Here is the code:

[6]https://github.com/bkiers/retrofit-oauth/tree/master/src/main/java/nl/bigo/retrofitoauth

```java
public class ServiceGenerator {

    private static final String BASE_URL = "https://api.github.com/"

    private static Retrofit.Builder builder =
        new Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create());

    private static OkHttpClient.Builder httpClient =
            new OkHttpClient.Builder();

    public static <S> S createService(Class<S> serviceClass) {
        builder.client(httpClient.build());
        Retrofit retrofit = builder.build();
        return retrofit.create(serviceClass);
    }
}
```

The `ServiceGenerator` class uses Retrofit's `Retrofit`-Builder to create a new REST client with the given API base url (`BASE_URL`). For example, GitHub's API base url is located at `https://api.github.com/` and you must update the provided example url with your own url to call your API instead of GitHub's.

The `serviceClass` defines the annotated class or interface for API requests. The following section shows the concrete usage of Retrofit and how to define an exemplary GitHub client.

### Why Is Everything Declared Static Within the Servicegenerator?

You might wonder why we use static fields and methods within the `ServiceGenerator` class. Actually, it has one simple reason: we want to use the same OkHttpClient throughout the app to just open one socket connection that handles all the request and responses including caching and many more. It's common practice to just use one OkHttpClient instances to reuse open socket connection. That means, we either need to inject the OkHttpClient to this class via dependency injection or use a static field. As you can see, we chose to use the static field. And because we use the OkHttpClient throughout this class, we need to make all fields and methods static.

## Retrofit in Use

Ok, let's use an example and define a REST client to request data from GitHub. First, we have to create an interface and define the required methods.

## GitHub Client

We'll use the already defined `GitHubService` interface from above that has only one method to request the repositories for a given user. Remember that we're replacing the path parameter placeholder (`{user}`) with the actual value of `user` when calling this method.

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

The interface above defines a `GitHubRepo` class. This class includes the required object properties to map the response data. For illustration purposes, we just define the repository's `id` and `name` properties. GitHub's API response for this endpoint has a lot more data, but is sufficient to show you how things work.

```java
public class GitHubRepo {
    private int id;
    private String name;

    public GitHubRepo() {}

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

With regard to the previous mentioned JSON mapping: the created `GitHubService` defines a method named `reposForUser` with the return type `List<GitHubRepo>`. The `List<GitHubRepo>` is wrapped into a `Call`. We'll cover the `Call` within the next chapter. For now it's ok to know that you need wrap your response into a call object.

If an appropriate response converter is present, Retrofit ensures that the server's JSON response gets mapped correctly to Java objects (assuming that the JSON data matches the given Java class).

## API Example Request

The snippet below illustrates the usage of the `ServiceGenerator` to instantiate your client, specifically the `GitHubService`, and the method call to get a user's repositories via that client. This snippet is a modified version of provided Retrofit github-client example[7].

```java
@Override
protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // Change base url to GitHub API
    ServiceGenerator.changeApiBaseUrl("https://api.github.com/");

    // Create a simple REST adapter which points to GitHub’s API
    GitHubService service =
            ServiceGenerator.createService(GitHubService.class);

    // Fetch and print a list of repositories for user “fs-opensource”
    Call<List<GitHubRepo>> call = service.reposForUser("fs-opensource");
    call.enqueue(new Callback<List<GitHubRepo>>() {
        @Override
        public void onResponse(
            public void onResponse(Call<List<GitHubRepo>> call,
                                   Response<List<GitHubRepo>> response) {
            if (response.isSuccess()) {
                for (GitHubRepo repo : response.body()) {
                        Log.d("Repo: ",
                                repo.getName() + " (ID: " + repo.getId() + ")");
                }
            } else {
                    Log.e("Request failed: ",
                            "Cannot request GitHub repositories");
            }
        }

        @Override
        public void onFailure(Call<List<GitHubRepo>> call, Throwable t) {
            Log.e("Error fetching repos", t.getMessage());
        }
    });
}
```

You’ve got a first impression of Retrofit and know how to define an interface which represents your API endpoints on client side. Besides that, you know how to create the API client with the help of Retrofit’s `Retrofit` class and how to create a generic `ServiceGenerator` for static service creation.

[7] https://github.com/square/retrofit/blob/master/samples/src/main/java/com/example/retrofit/SimpleService.java

We’ll update the `ServiceGenerator` in the following chapters within this book with examples for authentication.

The next chapter shows you how to define and manipulate requests with Retrofit.

## Chapter Summary

You’ve mastered your first steps to become a proficient developer using Retrofit. Within this chapter, you got an overview on Retrofit and we walked through a practice related example. You should have learned

- [x] What is Retrofit
- [x] Why use Retrofit
- [x] How to define a service interface representing an API endpoint
- [x] How to create an instance of the service interface
- [x] How to execute an API request

The next chapter is all about requests. We’ll guide you through the setup of Retrofit to receive and send data with your request.

# Chapter 2 — Requests

Retrofit offers a variety of capabilities to customize requests and their parameters to your needs. With the given functionality, you can adjust as you wish using annotations and placeholders and Retrofit handles everything for you.

Within this chapter, we dive into the manipulation of your requests. That means, how to correctly create your desired request url, define a dynamic url for selected endpoints, use path and query parameters, add custom headers. Additionally, Retrofit provides the functionality for synchronous and asynchronous requests. We take a look at how to perform both types. Last but not least, we go into detail on how to send data within HTTP's request body using either form-urlencoded data or a given Java object.

We already know Retrofit's annotations from the previous Quick Start chapter. However, let's quickly refresh how to declare an API on Android with annotations.

## API Declaration With Retrofit

This section is a short recap to remind you about this essential part of Retrofit: using annotations on Java interface methods to describe individual API endpoints and ultimately, the handling of requests. One of the first things you want to define is the HTTP request method like `GET`, `POST`, `PUT`, `DELETE`, and so on. Retrofit provides an annotation for each of these request methods. Additionally, you need to add the relative resource endpoint url to the annotation. You've the ability to use dynamic urls for individual requests, we'll touch this topic within the next section. For now, let's have a look at a common example defining the endpoint url. An example will clarify all theory:

```java
public interface TaskService {
    @GET("relative/url/path") // <-- this is the important part
    Call<Void> method();
}
```

More specifically, the definition above gets translated by Retrofit to a `GET` request with the relative url of `relative/url/path` to the resource endpoint. Retrofit's `Retrofit` class integrates the base url and the defined resource url within the `GET` annotation gets concatenated with the base url.

The following section describes in detail how Retrofit composes the final request url out of both string values.

## Url Handling, Resolution and Parsing

If you worked with Retrofit 1, you know the rudimentary behavior in regard to url handling. With Retrofit 2, there's a kind of fundamental shift in how to think and apply your urls including any type of parameter.

### Url Handling Introduction

In Retrofit 2, all your urls are resolved using the `HttpUrl` class that's brought to you by OkHttp. It's an awesome class that does its job very well. Nonetheless, it introduces changes in the way you need to handle all your urls within your app: base and endpoint urls as well as any dynamic url defined for a specific request.

**Remember:** your urls in Retrofit are handled like links on web pages: `<a href="…">…</a>`.

#### `baseUrl` Resolution

Using Retrofit, you're requesting a specific API that always has the same base address. This base address shares the same scheme and host and you can define it in a single place (using `Retrofit.Builder()`) and change it there if necessary without touching every endpoint in your app.

```
Retrofit.Builder builder = new Retrofit.Builder()
        .baseUrl("https://your.base.url/api/");
```

The base url is used for every request and any endpoint value (like `@GET`, `@POST`, etc.) gets resolved against this address. You should **always end your base url with a trailing slash**: `/`. And of course we tell you why: each endpoint definition with a relative path address will resolve correctly, because it appends itself to the base url that already defines or includes path parameters.

Let's look at an example to get things straight:

```
# Good Practice
base url: https://futurestud.io/api/
endpoint: my/endpoint
Result:   https://futurestud.io/api/my/endpoint

# Bad Practice
base url: https://futurestud.io/api
endpoint: /my/endpoint
Result:   https://futurestud.io/my/endpoint
```

The example above illustrates what happens if you don't end your base url on a trailing slash. The `api` path parameter will be omitted and removed from the resulting request url.

Actually, Retrofit helps you out if you pass a base url without trailing slash. It will throw an exception and tell you that your base url needs to end on a slash.

## Absolute Urls

You've already seen in the *bad practice* from above, that you can pass absolute urls to your endpoint urls. Nonetheless, this technique might be necessary in your app to call the appropriate endpoints. Over time, your backend will release a new API version. Depending on the type of versioning, let's assume your backend developer chose to version the API within the url. You need to bump your base url from `v2` to `v3`. At this point, you'll have to take care of all the breaking changes introduced by API v3. To rely on selected v2 endpoints, you can use an absolute endpoint url to directly specify an API version.

```
# Example 1
base url: https://futurestud.io/api/v3/
endpoint: my/endpoint
Result:   https://futurestud.io/api/v3/my/endpoint

# Example 2
base url: https://futurestud.io/api/v3/
endpoint: /api/v2/another/endpoint
Result:   https://futurestud.io/api/v2/another/endpoint
```

In situations when you change your base url, you're automatically upgrading all endpoints to use the new url and request against it. As you can see, `Example 1` works like expected and just appends the endpoint url to the base url for the API call against v3. If you don't expect a breaking change for the v3 endpoint, this is the most convenient way to jump on the upgrade train.

`Example 2` illustrates the case, that you're upgrading your base url to v3 and still use the API v2 for a selected endpoint. This might be caused due to huge upgrading efforts on your client side and you still want to profit from all the other benefits that come with API v3 endpoints.

## Dynamic Urls for Requests

Retrofit finally comes with an additional annotation for dynamic urls. What was a hack-around in Retrofit 1 is now just a single `@Url` annotation for the endpoint declaration. This section will show you how to make use of dynamic endpoint urls for individual requests.

## Use-Case Scenarios

There might not directly come a use case to your mind that requires the definition of the endpoint url on the fly. We can give you two examples to illustrate real world scenarios.

1. **Profile Pictures**: if your app allows users to upload their personalized profile picture, you might store them on different locations like your own server, Amazon S3 or maybe you also allow the linking of Gravatar.
2. **File Downloads**: Files can be stored at various locations and you want/need to be flexible to download them from any resource path.

Even though your Android app didn't have any of the proposed functionality, you're now in the appropriate mindset to imagine examples where dynamic urls are a good fit to make use of.

## How to Use Dynamic Urls

Actually, it only requires you to add a single String parameter annotated with `@Url` in your endpoint definition. Short code says more than thousand words :)

```
public interface UserService {
    @GET
    public Call<ResponseBody> profilePicture(@Url String url);
}
```

As you can see, you'll leave the `@GET` annotation without the endpoint url and add the `@Url` to the method itself. That's it!

## How Urls Resolve Against Your Base Url

That's another interesting part and needs attention: how will the dynamic url be resolved against the defined base url? Retrofit uses OkHttp's `HttpUrl` and resolves each endpoint url like a link on any website (`<a href="">…</a>`).

Let's touch some scenarios and at first look at a url that points to a profile picture stored on Amazon S3. For this example, we'll use the `UserService` from above. Within the following snippet, we define a base url and the actual call of the `profilePicture` method uses a completely different one.

```java
Retrofit retrofit = Retrofit.Builder()
    .baseUrl("https://your.api.url/");
    .build();

UserService service = retrofit.create(UserService.class);
service.profilePicture("https://s3.amazon.com/my/profile-picture/path");

// request url results in:
// https://s3.amazon.com/my/profile-picture/path
```

Because you set a completely different host including a scheme (`https://s3.amazon.com` vs. `https://your.api.url`), the `HttpUrl` from OkHttp will resolve it to the dynamic one.

Another example: this time we point the dynamic url for our profile picture to the same server as we've already defined as the base url.

```java
Retrofit retrofit = Retrofit.Builder()
    .baseUrl("https://your.api.url/");
    .build();

UserService service = retrofit.create(UserService.class);
service.profilePicture("my/profile-picture/path");

// request url results in:
// https://your.api.url/my/profile-picture/path
```

This time, the final request url gets resolved to a concatenation of the base url and our dynamically defined endpoint url. `HttpUrl` will recognizes that we didn't specify a scheme and host and therefore appends the endpoint url to the base url.

A third example: let's assume the backend developers pushed an update to production that bumps the API's base url to version 2.

```java
Retrofit retrofit = Retrofit.Builder()
    .baseUrl("https://your.api.url/v2/");
    .build();

UserService service = retrofit.create(UserService.class);
service.profilePicture("/my/profile-picture/path");

// request url results in:
// https://your.api.url/profile-picture/path
```

The difference between the second and third examples is this: we've added `v2/` to the base url and start the endpoint url with a leading `/`. Actually, this will result in the same final request url, because endpoint urls that start with a leading slash will be appended to only the base url's host. Everything that comes after the host url will be omitted when using leading slashes for endpoint urls. You can fix your problem by removing the leading `/` from your endpoint.

## Dynamic Urls or Passing a Full Url

You've the ability to pass a given url to an endpoint which then is used for the request. When using a dynamic url, Retrofit is very flexible and can keep your scheme from the base url. That means, if you've defined your base url with `https` and you want to make sure that all other requests in your app are also using `https`, you can just start your request urls with a doubled slash `//`. This is common practice in the web to avoid a browser warning on using secure and insecure resources on the same page.

```
# Example 3 — completely different url
base url: http://futurestud.io/api/
endpoint: https://api.futurestud.io/
Result:   https://api.futurestud.io/

# Example 4 — Keep the base url's scheme
base url: https://futurestud.io/api/
endpoint: //api.futurestud.io/
Result:   https://api.futurestud.io/

# Example 5 — Keep the base url's scheme
base url: http://futurestud.io/api/
endpoint: //api.github.com
Result:   http://api.github.com
```

`Example 3` shows you the replacement of the base url when using a completely different one. This example is useful when requesting files or images that have different locations, like some files are on your own server and others are stored on Amazon's S3. Assuming, that you're only receiving the location of the user profile picture as an url within the profile, with Retrofit you're able to pass the complete url for your requesting endpoint.

As already mentioned, you can keep the base url's scheme. In `Example 4`, we're not using the path segment for the API, but instead a subdomain. We still want to keep the previously defined scheme and therefore just pass the full url with leading `//`.

`Example 5` uses the same scheme as the defined base url, but replaces the host and path segments with the given endpoint url.

## Query Parameter

When working with APIs, you often want to filter resources down to a specific subset. A common use case is the selection of one element identified by a unique id value. In the following, you'll learn how to perform requests with single and multiple query parameters (those can have the same name).

### Query Parameter — Introduction

Query parameters are a common way to request filtered data from servers. Frequent examples of query parameters are requests to select specific elements, order the result set or use pagination to query the result data.

You can specify static (hard coded) or dynamic query parameters within the resource endpoint url. Let's use the example below which **always** requests the task with `id=123` from an example API.

```
public interface TaskService {
    @GET("tasks?id=123")
    Call<Task> getTask123();
}
```

Our imaginary API will return the data for a single task with `id=123`. The example above is quite uncommon in real world situations. Usually you need a more dynamic behavior for your request with changing values for given query parameters.

The Retrofit method definition for query parameters is straight forward. You add the `@Query("key")` annotation before your method parameter. The `key` value within the `@Query` annotation defines the parameter name within the url. Retrofit automatically adds those parameters to your request url.

```
public interface TaskService {
    @GET("tasks")
    Call<Task> getTask(
            @Query("id") long taskId
    );
}
```

The method `getTask(…)` requires the parameter `taskId`. This parameter will be mapped by Retrofit to a query parameter with the name specified within the `@Query()` annotation. In this case, the parameter name is `id` which will result in a request resource url like:

```
tasks?id=<taskId>
```

Of course, you can add more than one url parameter to your request. We'll cover that scenario in the following paragraph.

## Multiple Query Parameters

We already know how to define single static and dynamic query parameters. Now we're going to add multiple query parameters. You can probably guess the way to go: annotate all desired method parameters with the `@Query(…)` annotation and Retrofit does the job for you.

```
public interface TaskService {
    @GET("tasks")
    Call<List<Task>> getTasks(
            @Query("id") long taskId,
            @Query("order") String order,
            @Query("page") int page
    );
}
```

The resulting resource url of the request looks like:

```
tasks?id=<taskId>&order=<order>&page=<page>
```

Adding every possible query parameter to your method definition can cause a lot headache. Every time when calling this method you need to pass all the given parameters, even though their values doesn't matter. The following shows you how to avoid passing all method parameters.

## Multiple Query Parameter Using QueryMap

Retrofit uses annotations to translate defined keys and values into appropriate format. Using the `@Query("key") String value` annotation will add a query parameter with name `key` and the respective string value to the request url (of course you can use other types than string :)).

Actually, there are APIs with endpoints allowing you to pass (optionally) multiple query parameters. You want to avoid a service method declaration like the one below with "endless" options for request parameters:

```
public interface TaskService() {
    @GET("tasks")
    Call<List<Task>> getTasks(
            @Query("id") long taskId,
            @Query("order") String order,
            @Query("page") int page,
            @Query("owner") String owner
    );
}
```

You could call the `getTasks` service method with `null` values for each of the parameters to make them optional. Retrofit will ignore `null` values and don't map them as query parameters. However, there's a better solution to work with complex API endpoints having various options for query parameters. Don't worry, Retrofit got you covered!

## How to Use QueryMap

The `@QueryMap` annotation expects a standard Java `Map` implementation with key-value-pairs of type `String`. Each key which has a non-null value will be mapped and you can dynamically add your desired query parameters.

```
public interface TaskService() {
    @GET("tasks")
    Call<List<Task>> getTasks(
        @QueryMap Map<String, String> options
    );
}
```

If you want to request the tasks for `Marcus` from page `2`, you only need to add the map entries for `page` and `owner`. There's no need to worry about the other available options (`taskId`, `order`).

The resulting url with the assumed example (page `2` of Marcus):

```
tasks?page=2&owner=Marcus
```

## QueryMap Options

The `@QueryMap` annotation has an option field for encoding:

- `encoded`: can be either `true` or `false`; default is `false`

You can set the value for `encoding` like this:

```java
public interface NewsService() {
    @GET("tasks")
    Call<List<Task>> getTasks(
        @QueryMap(encoded=true) Map<String, String> options);
    );
}
```

Enabling `encoding` will encode individual characters before appended to the request url. Using the key `author` with value `marcus-poehls` will result in the following encoded url: `author=marcus%2Dpoehls`.

Setting `encoding` to `false` (by default), the url for the example above would be `author=marcus-poehls`.

There's another use case where Retrofit's query map would be impractical: adding multiple query parameters with the same key. The next section shows you how to add multiple query parameters with the same name.

## Multiple Query Parameters with the Same Name

Some use cases require to pass multiple query parameters with the same name. In regard to the already introduced example of requesting tasks from the imaginary API, we could extend the query parameter to accept a list with multiple task ids. The expected result is the corresponding set of tasks with the given task ids.

The request url we want to create should look like:

```
http://api.example.com/tasks?id=1&id=2&id=3
```

Retrofit executes requests with multiple query parameters of the same name by providing a list for the `id` query parameter. Then, Retrofit concatenates the given list to multiple query parameters of same name.

```java
public interface TaskService {
    @GET("tasks")
    List<Task> getTask(
            @Query("id") List<Long> taskIds
    );
}
```

Given a list of `taskIds=[1,2,3]` the resulting request url looks like this:

```
http://api.example.com/tasks?id=1&id=2&id=3
```

Besides single query parameters (static or dynamic), we've covered the definition of multiple query parameters with the same name. There are situations where you don't want to send values for all defined query parameters. Within those situations, Retrofit got your back with optional query parameters.

## Optional Query Parameters

In most cases, the sorting option is not required to successfully execute a request. The backend generally provides a default sorting and you can just use the data as is. The important thing to know is how to use optional query parameter in your client.

Let's assume the following code snippet defines our API client on Android and we don't want to pass a `sort` query parameter on every request. Only if the user specifies a sorting style, we use it.

```java
public interface TaskService {
    @GET("tasks")
    Call<List<Task>> getTasks(
            @Query("sort") String order
    );
}
```

Depending on the API design, the `sort` parameter might be optional. If you don't want to pass it with the actual request, just pass **`null`** as the value for `order` during the method call.

```java
service.getTasks(null);
```

Retrofit skips `null` parameters and ignores them while assembling the request. Please keep in mind, that you can't pass `null` for primitive data types like `int`, `float`, `long`, etc. Instead, use their class representations `Integer`, `Float`, `Long`, etc. and the compiler won't be grumpy.

The following example illustrates the service method definition with two parameters for `sort` and `page`. We have to use `Integer` for the `page` parameter if we want it to be optional.

```java
public interface TaskService {
    @GET("tasks")
    Call<List<Task>> getTasks(
            @Query("sort") String order,
            @Query("page") Integer page
    );
}
```

Now you can pass `null` for both `order` and `page` if you don't want to request a specific order for the results from a given page.

```java
service.getTasks(null, null);
```

**Remember**: query parameters are a powerful way to request filtered data from an API. Use them intentionally to find exactly what you're looking for. Usually, computing on server side is much more effective and you don't need to apply any further computing on client-side if you're already receiving the desired results.

## Add Query Parameters to Every Request

Adding query parameters to single requests is straight forward. You're using the `@Query` annotation for your parameter declaration within the interface methods. This tells Retrofit to translate the provided query parameter name and value to the request and append the fields to the url. Recently, we've been asked how to add a query parameter to every request[8] using an interceptor.

### Add Query Parameters Using an Interceptor

If you've used Retrofit before, you're aware of the `@Query` annotation used to add query parameters for single requests. There are situations where you want to add the same query parameter to every request, just like adding an `Authorization` header passing the authentication token. If you're querying an API which accepts an `apikey` as a query parameter, it's valuable to use an interceptor instead of adding the parameter to every request method.

You can do that by adding a new request interceptor to the `OkHttpClient`. Retrofit has a direct dependency for OkHttp and we'll use it as our low-level networking client. We can intercept the actual request and manipulate its parameters, headers, body and much more. The common way to apply custom directives before sending the request is a user-defined `Interceptor` implementation. Within the interceptor, we've access to the request and can manipulate it as planned.

Intercept the actual request and get the `HttpUrl`. The http url is required to add query parameter, because it will change the previously generated request url by appending the query parameter's name and its value.

[8]https://futurestud.io/blog/android-basic-authentication-with-retrofit/#comment-2344452356

The following code illustrates the desired implementation using an OkHttp interceptor. Please don't worry if you're unfamiliar with interceptors yet. We'll describe the code snippet below and have a deeper look at interceptors later within this book.

```
OkHttpClient.Builder httpClient =
    new OkHttpClient.Builder();

httpClient.addInterceptor(new Interceptor() {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request original = chain.request();
        HttpUrl originalHttpUrl = original.httpUrl();

        HttpUrl url = originalHttpUrl.newBuilder()
                .addQueryParameter("apikey", "your-actual-api-key")
                .build();

        // Request customization: add request headers
        Request.Builder requestBuilder = original.newBuilder()
                .url(url)
                .method(original.method(), original.body());

        Request request = requestBuilder.build();
        return chain.proceed(request);
    }
});
```

Once you've the `HttpUrl` object, you can create a new builder based on the original http url object. The builder will let you add further query parameters using the `.addQueryParameter(key, val)` method. Once you've added your query parameter, use the `.build()` method to create the new `HttpUrl` object which is added to the request by using the `Request.Builder`. The code above builds the new request with the additional parameter based on the original request and just exchanges the url with the newly created one.

Every request that uses the OkHttp client from above will have the defined query parameter within the request url.

Query parameters aren't the only way to filter resources from an API. Another way to dynamically select endpoints or resources are path parameters. The following section shows you how to use them within Retrofit.

## Path Parameter

Besides the query parameter, you can customize your request url with path parameters. Path parameters are defined as placeholders within your resource endpoint url and get substituted with parameter values from your method definition. The placeholders within the endpoint url are defined by a string value surrounded by curly brackets, like `{my-path-placeholder}`. Additionally, you need to annotate a method parameter with the `@Path("my-path-placeholder")` annotation.

```
public interface TaskService {
    @GET("/tasks/{id}")
    Call<Task> getTask(
        @Path("id") long taskId
    );
}
```

The string value of your path parameter within the resource url (`{id}`) must match the defined value for the `@Path("id")` annotation.

Of course, you can combine query and path parameters within the same method definition. Just add your desired query parameter to the method signature and use the `@Query` or `@QueryMap` annotations for the additional parameter.

**Path Parameters for Endpoint Urls**

The parameter defined for the `@Path()` annotation requires the same name as defined in the endpoint url. Having an endpoint url like `@GET("tasks/{myUser}")` with method definition `Call<List<Task>> tasksForUser(@Path("user") String username);` won't map correctly, because `myUser != user`.

### Optional Path Parameter

Requesting data from an API can include user defined filters for specific data. A common scenario is to filter a single item from its resource collection by using a query or path parameter. Depending on the API implementation, you're probably using the same endpoint to request different data with given filter values. We'll show you how use optional path parameters to refine your request for single or multiple values.

### API Description

Let's assume we want to request a list of tasks from an API. The tasks endpoint is located at `/tasks` and the API implementation allows you to filter the request to a single task by passing an individual task id, `/tasks/<task-id>`. You'll receive the same response format: a list of tasks. If you specify the task id, you'll receive a list of tasks with just a single item.

## Optional Path Parameter

Related to the imaginary API described above, we can use the following interface to request a list of tasks and also to filter down to a single task item.

```
public interface TaskService {
        @GET("tasks/{taskId}")
        Call<List<Task>> getTasks(
           @Path("taskId") String taskId
        );
}
```

As you can see in the `TaskService` above, we've defined a path parameter `taskId` that will map its value appropriately.

The trick is this: you need to use an empty string parameter. Retrofit will map the empty string value properly as it would do with any serious value, but the result is an url without path parameters.

Look, the following urls are handled the same on server-side which allows us to reuse the endpoint for different requests:

```
# will be handled the same
https://your.api.url/tasks
https://your.api.url/tasks/
```

The following code snippets show you how to request the general list of tasks and how to filter down to a single task item.

```
// request the list of tasks
TaskService service =
    ServiceGenerator.createService(TaskService.class);
Call<List<Task>> voidCall = service.getTasks("");

List<Task> tasks = voidCall.execute().body();
```

By passing an empty String to the `getTasks` method, Retrofit (especially OkHttp's `HttpUrl` class) will "remove" the path parameter and just use the leading url part for the request.

The code example below illustrates the request to filter a single task using the same endpoint. Because we're now passing an actual value for the path parameter, the mapping applies and the request url contains the parameter value.

```java
// request a single task item
TaskService service =
    ServiceGenerator.createService(TaskService.class);
Call<List<Task>> voidCall = service.getTasks("task-id-1234");

// list of tasks with just one item
List<Task> task = voidCall.execute().body();
```

That's the trick to reuse an endpoint if the API returns data in a consistent schema.

## Attention

Actually, the shown trick can cause issues as well. If your path parameter is a segment in the middle of the url, passing an empty string will result in a wrong request url. Let's assume there's a second endpoint responding with a list of subtasks for a given task.

```java
public interface TaskService {
        @GET("tasks/{taskId}/subtasks")
        Call<List<Task>> getSubTasks(
           @Path("taskId") String taskId
        );
}
```

If we provide the value `task-123` for `taskId`, the desired request url will be composed correctly:

```
https://your.api.url/tasks/task-123/subtasks
```

In contrast, passing an empty value for `taskId` will result in the following url:

```
https://your.api.url/tasks//subtasks
```

As you can see, the API won't be able to find the subtasks, because the actual task id is missing.

## Don't Pass `Null` as Parameter Value

Retrofit doesn't allow you to pass `null` as a value for path parameters and if you do, it throws an `IllegalArgumentException`. That means, your app will crash at runtime! Be aware of this behavior with requests that involve a path parameter. Ensure stability by verifying that the path parameter values are always not `null`.

This part showed you how to reuse method definitions within your service interfaces in situations where the API responses are consistent across different endpoints. You can benefit from optional path parameters to request a collection or single items from a resource using the same endpoint definition.

Be aware, that this shortcut can't be applied to path parameters that are located in between parts of the url. Retrofit will remove path parameters with an empty value and the resulting request url probably doesn't match your desired endpoint.

Within the first sections, you've already learned how to describe your API endpoints on client-side and how to request specific data with the help of query and path parameters.

Finally, the following section will show you how to execute your request.

## Synchronous and Asynchronous Requests

Retrofit supports synchronous and asynchronous request execution out of the box. If you worked with Retrofit 1, you're familiar with the different method declarations within the service interface. Synchronous methods required a return type. In contrast, asynchronous methods required a generic `Callback` as the last method parameter.

Within Retrofit 2, there's no differentiation anymore for synchronous and asynchronous requests. Requests are now wrapped into a generic `Call` class using the desired response type. That means, you don't specify the request type (synchronous/asynchronous) within the service interface, but when ultimately executing the request using a `Call` instance.

### Synchronous Requests

Every request is wrapped into a `Call` object. When your service interface and calling the desired method, you'll receive a `Call` object. Having the call object, you need to use `call.execute()` to perform a synchronous request.

The example below yields a list of tasks as the response when executing the method `getTasks()`.

```
public interface TaskService {
    @GET("tasks")
    Call<List<Task>> getTasks();
}
```

The snippet above depicts the interface we're using to request a list of tasks. The corresponding code to execute the actual request is shown below:

```java
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<List<Task>> call = service.getTasks();
List<Task> tasks = call.execute().body();
```

At first, we create an instance of the `TaskService` using the introduced `ServiceGenerator` from chapter 1. Second, we instantiate the call object required to execute the request and ultimately receive the list of tasks as a response.

Synchronous methods executed on Android's main/UI thread will block any user action. That means the UI is blocked during request execution and no further interaction is possible for this period.

However, **synchronous methods are only blocking within the thread they get started**. Specifically, if you run a synchronous task on an asynchronous or background thread, the UI won't be affected at all. There are situations when synchronous requests on already asynchronous threads are required to achieve the solution. We'll make use of this behavior within the authentication chapter (chapter 4) to refresh an invalid access token.

**Don't execute synchronous methods on Android's Main/UI Thread**

Synchronous requests can be the reason of app crashes on Android equal or greater than 4.0. You're going to get an `android.os.NetworkOnMainThreadException` error if you're performing synchronous network operations.

Synchronous methods provide the ability to use the return value directly, because the operation blocks everything else during your network request.

For non-blocking UI, you need to handle the request execution in a separate thread by yourself. That means, you can still interact with the app itself while waiting for the response. However, Retrofit has the functionality for asynchronous requests natively and the following section will go into detail.

## Asynchronous Requests

In addition to synchronous calls, Retrofit supports asynchronous requests out of the box. As already described, you differentiate the request type by using the `Call` object. That means, we can use the same interface definition to request a list of tasks from our example API.

```java
public interface TaskService {
    @GET("tasks")
    Call<List<Task>> getTasks();
}
```

To execute a request asynchronously, use the `call.enqueue()` method. Behind the scenes, Retrofit handles the execution on a separate thread so you're not interfering with your main thread or even blocking the UI.

Using asynchronous requests requires you to implement a `Callback` with two callback methods: `onResponse()` and `onFailure()`. The callback implementation defines what should be done once the request finishes. The following code snippet illustrates an exemplary implementation.

```
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<List<Task>> call = service.getTasks();
call.enqueue(new Callback<List<Task>>() {
    @Override
    public void onResponse(Call<List<Task>> call,
                           Response<List<Task>> response) {
        // response.isSuccessful() is true if the HTTP response status code is 2\
xx
        if (response.isSuccessful()) {
                // use the returned tasks here
        } else {
            // handle request errors yourself
            // depending on status code :)
            int statusCode = response.code();
        }
    }

    @Override
    public void onFailure(Call<List<Task>> call, Throwable t) {
        // handle execution failures like no internet connectivity
    }
});
```

The `onResponse()` method is called for any response that can be handled correctly even in failure situations when the status code isn't in the range of `200-299`. The `Response` class has a convenience method `isSuccessful()` to check whether the request was handled successfully (returning status code `2xx`) and you can use the response object for further processing. If the status code isn't `2xx`, you need to handle errors yourself. If you're expecting a response body in failure situations (like an error message), you can parse the JSON with the error body yourself by using the `errorBody()` method of the `ResponseBody` class. We'll get back to error handling in a later chapter.

## Get Raw HTTP Response

If you need the raw HTTP response object, you need to make use of OkHttp's `ResponseBody` class. Define it as the return type for the typed `Call` class within your service method definition.

```java
public interface TaskService {
    @GET("tasks")
    Call<ResponseBody> getTasks();
}
```

**Attention**: Retrofit doesn't allow you to define any other return type to receive the raw response. It'll throw an exception if the response was already converted to one or many Java objects and you try to access the raw body.

```java
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<ResponseBody> call = service.getTasks();
call.enqueue(new Callback<ResponseBody>() {
    @Override
    public void onResponse(Call<ResponseBody> call,
                           Response<ResponseBody> response) {
        // use the response body
        ResponseBody body = response.body();
    }

    @Override
    public void onFailure(Call<ResponseBody> call, Throwable t) {
        // handle execution failures like no internet connectivity
    }
});
```

The code snippet above just illustrates the asynchronous request that passes through the raw response to your callback. There's multiple situations where you need to access the response without applying a response converter and that's a way to go.

For now, we've touched the topics on how to request data from an API. The following will show you how to send data with your requests.

## Send Objects in Request Body

Of course, Retrofit offers the ability to send data to your desired API within the request body. You need to use the `@Body` annotation for your method parameter to tag it as your request body.

```java
public interface TaskService {
    @POST("tasks")
    Call<Task> createTask(@Body Task task);
}
```

Retrofit handles the serialization of your Java object to any other representation, like JSON, using the specified converter. Remember that you need to manually add a converter to the `Retrofit` instance. If you did that, Retrofit handles all the steps to create JSON from your given object and adds it as the body data to the request.

Please keep in mind, that you can't use the `@Body` annotation for `GET` requests!

## Example

Let's have a look at an example. Most of the examples within this book are based on an exemplary API around tasks. This example assumes our `Task` class is quite simple and has only 2 fields (to keep complexity to a minimum).

```java
public class Task {
    private int position;
    private String title;

    public Task() {}

    public Task(int position, String title) {
        this.position = position;
        this.title = title;
    }
}
```

Instantiating a new `Task` object fills its properties with values for `position` and `title`. Further, when passing the object to the service method, the object fields and values will be converted to JSON before sending the request.

The following snippet illustrates the example usage of both classes (`TaskService` and `Task`) from above. Also, we'll make use of the `ServiceGenerator` from the first chapter.

```
Task task = new Task(10, "my task title");

TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<Task> call = service.createTask(task);
call.enqueue(new Callback<Task>() {
    @Override
    public void onResponse(…) { }

    @Override
    public void onFailure(…) { }
});
```

Once we execute the request using the `call` instance, Retrofit kicks of the routines to assemble the request and that includes the mapping or conversion of parameters or data assigned with the request. In our scenario, we only want the `task` object in JSON representation. The JSON within the request body of `task` will look like this (uncompressed):

```
{
        "position": 10,
        "title": "my task title"
}
```

Within this section, we've instantiated a task object by passing all values within the contstructor. The following section will show you how to optimize your code for constant, default or calculated values sent within the request body.

## Constant, Default and Logic Values for POST and PUT Requests

This section explains how to add constant, default and logic (which are computed for some input) values to the HTTP request body for POST and PUT requests. Sounds complicated? We promise, it won't be!

As you've come so far, you already know how to create services and execute requests. If you feel uncomfortable reading these words, start over with this chapter to recap the fundamentals.

### Adding a Request for a Feedback Function

Let's assume your boss gave you the task to implement a feedback function to your Android app. The feedback function should allow the user to give some text input along with some general device data, which is collected automatically. Your backend developer has already done the work and described the endpoint to you. The endpoint expects the following:

```
URL: /feedback
Method: POST
Request Body Params (Required):

- osName=[String]
- osVersion=[Integer]
- device=[String]
- message=[String]
- userIsATalker=[boolean]

Response: 204 (Empty Response Body)
```

The first three values are data about the device the user is sending his feedback from. The fourth one is his actual message. The last one is a flag, which signals if the message is especially long.

## Simple Approach

The first attempt might be to transfer the REST API documentation over to our Retrofit service declaration:

```
@FormUrlEncoded
@POST("/feedback")
Call<ResponseBody> sendFeedbackSimple(
    @Field("osName") String osName,
    @Field("osVersion") int osVersion,
    @Field("device") String device,
    @Field("message") String message,
    @Field("userIsATalker") Boolean userIsATalker);
```

If you're unfamiliar with the `@FormUrlEncoded` and `@Field` annotations, don't worry. We'll explain them in detail in the next section. For now it's sufficient to understand what the data looks like that will be sent.

Next, you'd hook the `sendFeedbackSimple` method up to the onclick method of the submit button of the feedback form. There you'd gather the data, compute the logic value and call the service method with all of the data's values:

```java
private void sendFeedbackFormSimple(@NonNull String message) {
    // create the service to make the call, see first Retrofit blog post
    FeedbackService taskService =
        ServiceGenerator.create(FeedbackService.class);

        // create flag if message is especially long
    boolean userIsATalker = (message.length() > 200);

    Call<ResponseBody> call = taskService.sendFeedbackSimple(
            "Android",
            android.os.Build.VERSION.SDK_INT,
            Build.MODEL,
            message,
            userIsATalker
    );

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

That's it! The programmer in you is satisfied with the result, but you can hear your software engineering professor scream in agony. As you can see above in the method declaration of `sendFeedbackFormSimple()`, the only true variable is the message of the user. The `osName` variable will not change ever and is actually a constant (the Android app will always have `Android` as operating system). Next, the `osVersion` and `device` are default strings depending on the device model. The `userIsATalker` is a logic value, which is calculated the same way every time.

In this case, it's not a big issue, because you only have one feedback form in your app. But what if this would be an endpoint you call from multiple places in your app (for example updating the user profile)? Then you'd have a bunch of duplicated code. Of course, you could introduce some constants and a helper method, which keeps the logic in one place, but there is a cleaner way. In the next section we'll show you how you can create a `POST`/`PUT` request with just the one true parameter, while the rest gets set automatically.

## Advanced Approach With Passing the Only True Variable

First, we've to change the service declaration. The request now is made with a Java object (Remember: Retrofit will automatically serialize it):

```
@POST("/feedback")
Call<ResponseBody> sendFeedbackConstant(@Body UserFeedback feedbackObject);
```

The parameter is an instance of the `UserFeedback` class, which we haven't shown you yet. It's where the magic happens:

```
public class UserFeedback {

    private String osName = "Android";
    private int osVersion = android.os.Build.VERSION.SDK_INT;
    private String device = Build.MODEL;
    private String message;
    private boolean userIsATalker;

    public UserFeedback(String message) {
        this.message = message;
        this.userIsATalker = (message.length() > 200);
    }

        // getters & setters
        // ...
}
```

The constructor for this class only contains the true variable `message`, everything else gets set or computed automatically! Due to the reason we're passing the entire object, all the values will be sent.

This makes the request call back in our UI code much leaner as well:

```
private void sendFeedbackFormAdvanced(@NonNull String message) {
    FeedbackService taskService =
        ServiceGenerator.create(FeedbackService.class);

    Call<ResponseBody> call =
        taskService.sendFeedbackConstant(new UserFeedback(message));

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

Even if the app has multiple locations where this request is made, all the constant, default and logic values are computed in one single place. The request calls only pass the single variable. This makes your code much more robust against future changes (and easier to read)!

Sending data within the request body using the `@Body` annotation is just one of many opportunities. You can also assemble your request body using form-urlencoded data.

## Send Data Form-Urlencoded

Multiple methods exist to send data to your server. Besides using the `@Body` annotation, there's another type available to send data to an API or server with your request: form-urlencoded. In the following, we show you how to annotate your service interface and execute form encoded requests using Retrofit.

### Form Encoded Requests

Performing form-urlencoded requests using Retrofit is sort of straight forward. It's just another Retrofit annotation, which will adjust the proper mime type of your request automatically to `application/x-www-form-urlencoded`. The following interface definition will show you how to annotate your service interface for form-encoded requests. We'll describe the magic behind the curtain below the code snippets :)

```java
public interface TaskService {
    @FormUrlEncoded
    @POST("tasks")
    Call<Task> createTask(@Field("title") String title);
}
```

We're using the `TaskInterface` to illustrate the code. The important part is the `@FormUrlEncoded` annotation. Please be aware of the fact, that you can't use this annotation for `GET` requests. Form encoded requests are intended to send data to the server!

Additionally, you need to use the `@Field` annotation for the parameters which you're going to send with your request. Place your desired `key` inside the `@Field("key")` annotation to define the parameter name. Further, add the type of your value as the method parameter. If you don't use `String`, Retrofit will create a string value using Java's `String.valueOf(yourObject)` method.

Let's have a look at an example to illustrate all the theory! Using the following service call

```java
service.createTask("Research Retrofit form encoded requests");
```

results in the following form encoded request body:

```
title=Research+Retrofit+form+encoded+requests
```

**Of course, you can have multiple parameters for your request. Just add further `@Field` annotations with the desired type.**

## Form Encoded Requests Using an Array of Values

The example above describes the usage of the `@Field` annotation only for a single value. If you don't use String as the object type, Retrofit will do the job to parse a string value from the given object.

However, you can pass an array of values using the same `key` for your form encoded requests.

```
public interface TaskService {
    @FormUrlEncoded
    @POST("tasks")
    Call<List<Task>> createTasks(@Field("title") List<String> titles);
}
```

As you can see, the only difference to the previously described service interface is the list of titles as the parameter. Retrofit will map a given list titles to the respective key `title`.

Alright, time to touch another example to get an impression of how the final request will look like.

```
List<String> titles = new ArrayList<>();
titles.add("Research Retrofit");
titles.add("Retrofit Form Encoded")

service.createTasks(titles);
```

results in the following form encoded request body:

```
title=Research+Retrofit&title=Retrofit+Form+Encoded
```

Each item within the list of task titles will be mapped to a key-value-pair by Retrofit. The `title` key will be the same for every pair. The key-value-pairs are separated by an ampersand (`&`) and the values are separated from their keys by an equal symbol `=`.

## Field Options

The `@Field` annotation has an option field for encoding:

- `encoded`: can be either `true` or `false`; default is `false`

The `encoded` option defines whether your provided key-value-pair is already url encoded. To specify the encoded option value, you need to pass it within the `@Field` annotation. The example below illustrates a code examples and sets the encoded option to `true`.

```
public interface TaskService {
    @FormUrlEncoded
    @POST("tasks")
    Call<List<Task>> createTask(@Field(value = "title", encoded = true) String t\
itle);
}
```

Let's assume we use the key `title` with value `Research Retrofit` will result in the following encoded url: `title=Research%20Retrofit`. Setting `encoded` to `false` (by default), the url for the example above would be `title=Research Retrofit`.

## Send Data Form-Urlencoded Using FieldMap

This part continues on how to use form-urlencoded requests with Retrofit to transmit data with your request. If the API you're going to request accepts a simple JSON payload for processing, this approach is definitely worth a look.

### What Is `@Fieldmap` in Retrofit?

You're already familiar with Retrofit's annotation to map parameter values into appropriate format. There are multiple annotations for different kinds of situations, like adding query or path parameters, request payload using a given object or create the body from fields with form-urlencoded requests.

Let's shift focus to an example that makes things more approachable. Assuming that you have the option to update user data within your Android app, you want to call an API endpoint that takes an object of key-value-pairs.

We want to use a form-urlencoded request, because the API accepts a JSON object representing the fields that should be updated. You're tempted to define the API endpoint on client-side using the following interface definition.

```
public interface UserService {
    @FormUrlEncoded
    @PUT("user")
    Call<User> update(
            @Field("username") String username,
            @Field("name") String name,
            @Field("email") String email,
            @Field("homepage") String homepage,
            @Field("location") String location
    );
}
```

The `PUT` request requires values for multiple parameters like username, email, homepage, etc.

**The downside**: every time we want to send an update with the new user data, we have to provide a value for each parameter even though they didn't change. It blows up your code and imagine that the number of parameters doubles or even triples evolutionary! The length of the method call text will explode Android Studio's canvas.

Actually, there's a more elegant solution already integrated with Retrofit: `@FieldMap`.

## Form Encoded Requests Using FieldMap

In situations where you just need to send selected fields that should be updated for a given user, using Retrofit's `@FieldMap` is a lot more elegant. You can add the desired key-value-pairs to a standard Java `Map` implementation and they will be translated as you go.

```
public interface UserService {
    @FormUrlEncoded
    @PUT("user")
    Call<User> update(@FieldMap Map<String, String> fields);
}
```

Specifically, if you only want to update the username, there's no need to add any other field than the username. Your request payload only consists of the single field!

You'll find various applications to `@FieldMap` in your app. Of course, there's a dark side attached to this approach: you don't intuitively know which fields are allowed and what are their names. That requires some additional documentation within your code and if that's the only thing to be aware of, go for it!

## FieldMap Options

The `@FieldMap` annotation has an option field for encoding:

- `encoded`: can be either `true` or `false`; default is `false`

The example below outlines the code example to activate the encoded option.

```
@FieldMap(encoded = true) Map<String, String> fields
```

The `encoded` option defines whether each provided key-value-pair is already url encoded. To specify the encoded option value, you need to pass it within the `@FieldMap` annotation.

Let's look at an example. We want to update the `username` field to the new value `marcus-poehls`. With the default behavior (`encoded=false`), the key-value-pair results in `username=marcus-poehls`. Encoding enabled has the result `username=marcus%2Dpoehls`.

Form-urlencoded requests are convenient to create the desired request payload without defining a class representing the data. You can send selected fields with their values and don't blow up the request body with unnecessary or already known information.

### Form-Urlencoded vs. Query Parameter

When seeing the results for a form encoded request, you may ask yourself "What is the difference between form-urlencoded and query parameters?". In essence: the request type.

- form-urlencoded: `POST`
- query parameter: `GET`

Use form-urlencoded requests to send data to a server or API. The data is sent within the request body and not as an url parameter.

Query parameters are used when requesting data from an API or server using specific fields or filter.

## Add Request Header

Working with APIs doesn't just narrow down to request composition with query or path parameters and sending data within the request body. You also need to deal with HTTP headers for operations like cache adjustment or authentication.

The following subsections walk you through the definition of static and dynamic headers and how to add them to your requests.

### Define Custom Request Headers

Retrofit provides two options to define HTTP request header fields: **static** and **dynamic**. **Static** headers can't be changed for different requests. The header keys and respective values are fixed and initiated with the app startup.

In contrast, **dynamic** headers must be set for each request.

#### Static Request Header

The first option to add a static header is to define the header and its value for your API method as an annotation. The header gets automatically added by Retrofit for every request using this method. The annotation can be either a key-value-pair (as one string) or as a list of strings. Let's look at two examples to illustrate the both options:

```
public interface TaskService {
    @Headers("Cache-Control: max-age=640000")
    @GET("tasks")
    Call<List<Task>> getTasks();
}
```

The example above shows the key-value-definition for the static header. Further, you can pass a list of strings to the `@Headers` annotation and define multiple header fields at the same time.

```
public interface TaskService {
    @Headers({
            "Accept: application/vnd.yourapi.v1.full+json",
            "User-Agent: Your-App-Name"
    })
    @GET("tasks/{task_id}")
    Call<Task> getTask(@Path("task_id") long taskId);
}
```

There's another option to define static request headers: with an OkHttp interceptor. Before moving on with headers in interceptors, let's stay within the service interface and look at how to define dynamic headers.

### Dynamic Request Header

A more customizable approach are **dynamic headers**. A dynamic header is passed like a parameter to the method. The provided parameter value gets mapped by Retrofit to the request header before executing the request. Let's look at the code example:

```
public interface TaskService {
    @GET("tasks")
    Call<List<Task>> getTasks(@Header("Content-Range") String contentRange);
}
```

Define dynamic headers where you might pass different values for each request. The example illustrates the dynamic header with `Content-Range` definition[9].

**No Header Override**

Retrofit doesn't override header definitions with the same name. Every defined header gets added to the request, even if they duplicate themselves.

As promised, we're moving on from the service interface to OkHttp's request interceptors and show you how to use them to manage headers correctly.

[9]https://devcenter.heroku.com/articles/platform-api-reference#ranges

## Manage Request Headers in OkHttp Interceptor

You've already learned how to add custom request header using Retrofit. This part specifically touches the details on adding headers using OkHttp's request interceptor. Interceptors are a good way to statically change requests executed with the specific service client which has the interceptor assigned.

Adding HTTP request headers is a good practice to add information for API requests. A common example is authorization using the `Authorization` header field. If you need the header field including its value on almost every request, you can use an interceptor to add this piece of information. This way, you don't need to add the `@Header` annotation to every endpoint declaration.

OkHttp interceptors offer two ways to add header fields and values. You can either override existing headers with the same key or just add them without checking if there is another header key-value-pair already existing. We'll walk you through both of them in the following paragraphs.

### How to Override Headers

Intercepting HTTP requests with the help of OkHttp interceptors allows you to manipulate the actual request and apply your customization. The request builder has a `.header(key, val)` method which will add your defined header to the request. If there is already an existing header with the same `key` identifier, this method will override the previously defined value.

```java
OkHttpClient.Builder httpClient =
    new OkHttpClient.Builder();

httpClient.addInterceptor(new Interceptor() {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request original = chain.request();

        // Request customization: add request headers
        Request.Builder requestBuilder = original.newBuilder()
                .header("Authorization", "auth-value") // <-- important line
                .method(original.method(), original.body());

        Request request = requestBuilder.build();
        return chain.proceed(request);
    }
});
```

Retrofit, and especially OkHttp, allow you to add multiple headers with the same key. The `.header` method will replace all existing headers with the defined key identifier. Within the code snippet above, every `Authorization` header (if multiple have been defined already) will be updated and their previous value will be replaced with `auth-value`.

## How to Not Override Headers

There are situations where multiple headers with the same name are beneficial. Actually, we're only aware of one specific example from practice: the `Cache-Control` header. The HTTP RFC2616[10] specifies that multiple header values with the same name are allowed if they can be stated as a comma-separated list.

That means,

```
Cache-Control: no-cache
Cache-Control: no-store
```

is the same as

```
Cache-Control: no-cache, no-store
```

Using Retrofit and an OkHttp interceptor, you can add multiple request headers with the same key. The method you need to use is `.addHeader`.

The following example will add the `Cache-Control` headers from the example above:

```java
OkHttpClient.Builder httpClient =
    new OkHttpClient.Builder();

httpClient.addInterceptor(new Interceptor() {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request original = chain.request();

        // Request customization: add request headers
        Request.Builder requestBuilder = original.newBuilder()
                    .addHeader("Cache-Control", "no-cache")
                    .addHeader("Cache-Control", "no-store")
                    .method(original.method(), original.body());

        Request request = requestBuilder.build();
        return chain.proceed(request);
    }
});
```

[10]http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4.2

### Take Away

**Notice** the small but mighty difference:

- `.header(key, val)`: will override preexisting headers identified by `key`
- `.addHeader(key, val)`: will add the header and don't override preexisting ones

Retrofit simplifies header manipulation and allows to simply change them for separated requests when necessary. Make sure you're using the appropriate method to achieve your desired request results.

## Reuse, Analyze & Cancel Requests

Retrofit 2 fundamentally changes the way requests are made. Especially for developers, who are used to Retrofit 1.9 or earlier versions, the new `Call` class is something that takes time to get an understanding for. In this section, we'll show you some basic usage and features of this new mysterious class!

### Reuse of Call Objects

Actually, we'll use the familiar task API that we've touched for illustrations throughout this chapter. What kind of request it is, doesn't really matter. All the features we'll show you apply for all Retrofit requests.

#### One Request for Each Call Object

One of the first things you've to know about the `Call` class and its instances: each instance is special made for one and only one request. You cannot simply use the same object and call `execute()` or `enqueue()` again:

```
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<List<Task>> originalCall = service.getTasks();
Callback<List<Task>> taskCallback = new Callback<List<Task>>() { … };

// correct usage:
originalCall.enqueue(taskCallback);

// some other actions in between
// ...

// incorrect reuse:
// if you need to make the same request again,
// don't use the same originalCall again!
// it'll crash the app with a
// "java.lang.IllegalStateException: Already executed".
originalCall.enqueue(taskCallback); // <-- would crash the app
```

### Duplicated Call Objects

Instead, if you want to make the exact same request again, you can use the `clone()` method on the `Call` instance to create a copy of it. It'll contain the exact same settings and configuration. You can use the new copy to make an identical request to the server:

```
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<List<Task>> originalCall = service.getTasks();
Callback<List<Task>> taskCallback = new Callback<List<Task>>() { … };

// correct reuse:
Call<ResponseBody> newCall = originalCall.clone();
newCall.enqueue(taskCallback);
```

This can be very useful if you need to send the same request multiple times.

## Analyzing Requests With the Call Object

As you've already seen, Retrofit includes the `Call` instance to every callback method. Thus, if the request was successful or even if it failed, you still can access the original `Call` object. But the reason to include it is not so that you can reuse the `call` (hint: use `clone()`), it's that you can analyze the request that was made using `call.request()`. This will return the complete request data. Let's look at an example:

```
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<List<Task>> originalCall = service.getTasks();

originalCall.enqueue(new Callback<List<Task>>() {
    @Override
    public void onResponse(Call<List<Task>> call,
                           Response<List<Task>> response) {
        checkRequestContent(call.request());
    }

    @Override
    public void onFailure(Call<List<Task>> call, Throwable t) {
        checkRequestContent(call.request());
    }
});
```

Both callback methods are passing the request to a new helper method `checkRequestContent()`, which accesses the request headers, body and url:

```
private void checkRequestContent(Request request) {
    Headers requestHeaders = request.headers();
    RequestBody requestBody = request.body();
    HttpUrl requestUrl = request.url();

    // todo make decision depending on request content
}
```

This might be useful to you if you want to review the request, which was send to the server.

## Preview Requests

It's also good to know that you can utilize `call.request()` at any time, even if the request wasn't scheduled or made yet! It'll generate the request just like it would for a regular network call.

**Important**: the `call.request()` method does some significant computing, if the request wasn't executed yet. It's not recommended to get a preview of the data via `.request()` on Android's UI/Main thread!

## Cancel Requests

In this second part we still focus the new Retrofit `Call` class. If you haven't read the first part, where we look on how to reuse and analyze the request using the `Call` class, please check the previous section.

Here, we'll show you how to cancel requests and how to test if a request failed or was cancelled. Both can be very useful for app developers, who need to interact with APIs. If you're interested, keep reading!

### Cancel Requests

Let's use the following example code from that loads a list of tasks from an API. But the exact request doesn't matter, since this is applicable to any Retrofit request. We've added a new activity, which creates a new request and executes it just like we usually do:

```java
public class CallExampleActivity extends AppCompatActivity {

    public static final String TAG = "CallInstances";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_load_tasks);

        TaskService service =
            ServiceGenerator.createService(TaskService.class);

        Call<List<Task>> call = service.getTasks();

        call.enqueue(new Callback<List<Task>>() {
            @Override
            public void onResponse(Call<List<Task>> call,
                                   Response<List<Task>> response) {
                Log.d(TAG, "request success");
            }

            @Override
            public void onFailure(Call<List<Task>> call, Throwable t) {
                Log.e(TAG, "request failed");
            }
        });
    }

    // other methods
    // ...
}
```

Now let's assume the user clicked the "Abort" button or the app moved into a state where the request isn't necessary anymore: we need to cancel the request. In Retrofit, this can easily be done by using `call.cancel();`. This will either cancel the network request, if it's not already running, or not start the request at all. That's it! If we expand the relevant request code by a simple cancel action, it'd look like this:

```java
TaskService service =
    ServiceGenerator.createService(TaskService.class);

Call<List<Task>> call = service.getTasks();

call.enqueue(new Callback<List<Task>>() {
    @Override
    public void onResponse(Call<List<Task>> call,
                           Response<List<Task>> response) {
        Log.d(TAG, "request success");
    }

    @Override
    public void onFailure(Call<List<Task>> call, Throwable t) {
        Log.e(TAG, "request failed");
    }
});

// something happened, for example: user clicked cancel button
call.cancel();
```

## Check If Request Was Cancelled

If you cancel the request, Retrofit will classify this as a failed request and thus call the `onFailure()` callback in your request declaration. This callback is also used when there is no Internet connection or something went wrong on the network level. From the app perspective, these errors are quite a bit different. Thus, it's pretty helpful to know if a request was cancelled or if the device has no Internet connection.

Fortunately, the `Call` class offers a `isCanceled()` method to check whether the request was cancelled or not. With this little check you can distinguish various errors in your callback:

```java
new Callback<List<Task>>() {
    @Override
    public void onResponse(Call<List<Task>> call,
                           Response<List<Task>> response) {
        Log.d(TAG, "request success");
    }

    @Override
    public void onFailure(Call<List<Task>> call, Throwable t) {
        if (call.isCancelled()) {
            Log.e(TAG, "request was cancelled");
        } else {
            Log.e(TAG, "other larger issue, i.e. no network connection?");
        }
    }
}
```

Within the example above, we're leveraging the convenience method `isCanceled()` on the call instance to verify the failure. You should handle each situation appropriately. Depending on your app state, you might want to dismiss a loading indicator or show an error message to the user. Keep an eye on each situation and don't let the user become lost by not knowing what happens.

## Chapter Summary

This chapter gave you a detailed overview on how to customize requests to your needs. We covered query and path parameters, how to send data with your requests using the request body, how to use synchronous and asynchronous requests, how to define HTTP headers generally or specifically for requests and finally the reuse, analysis and cancellation of requests.

You've learned a lot! Let's recap in short what you should take away from this chapter.

- [x] How to use single and multiple query parameter
- [x] How to make use of path parameter
- [x] Send data with your request
- [x] Manage request headers
- [x] Reuse, analyze and cancel requests

The next chapter is in detail about managing responses and how to apply response converters for various data types (JSON, XML, protocol buffers, etc.) and map the received data to Java objects.

# Chapter 3 — Data Mapping with Retrofit Converters

Typically your requests are followed up with a response and that's usually the reason why we do all the request composition and manipulation. We want to display data in our Android app to be sure that the server processed our requested status update correctly or the delete operation finished successfully.

A common representation of the data received from the server is the javascript object notation: JSON. HTTP's response body contains any information and Retrofits parses the data and maps it into a defined Java class. This process can be kind of tricky, especially when working with custom formatted dates, nested objects wrapped in lists, and so on …

In order to start things off, this chapter shows you how to integrate a JSON converter. Next, you'll see how to integrate an XML response converter with Retrofit and how to define your Java objects to map to the XML data. Furthermore, we'll give an overview over all the other standard format converters Retrofit provides. Finally, we'll conclude the chapter with creating our own custom response converter, in case you're using some fancy custom data format.

## Integrate a JSON Converter

Let's start with the most common format: JSON. As we've mentioned in the Quick Start chapter, Retrofit doesn't ship with an integrated JSON converter anymore. We need to include the converter manually. For those of you, who already have worked through the first chapter, you've edited your `build.gradle`. In case you haven't done that yet, it's the time to do it now:

```
dependencies {
    // Retrofit & OkHttp
    compile 'com.squareup.retrofit2:retrofit:2.0.0'

    // JSON Converter
    compile 'com.squareup.retrofit2:converter-gson:2.0.0'
}
```

Google's Gson[11] is a very popular JSON converter. Almost every real-world JSON mapping can be done with the help of Gson. Once you've added the Gradle dependency to your project you've to activate it on your Retrofit instance. Only then Retrofit will consider using the converter.

[11]https://github.com/google/gson

```java
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com")
        .addConverterFactory(GsonConverterFactory.create())
        .build();

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

The converter dependency provides a `GsonConverterFactory` class. Call the created instance of that class (done via `create()`) to your Retrofit instance with the method `addConverterFactory()`. After you've added the converter to Retrofit, it'll automatically take care of mapping the JSON data to your Java objects, as you've seen in the previous chapter. Of course, this works for both directions: sending and receiving data.

The advantage of Gson is that it usually needs no setup in your Java classes. Nevertheless, there are notations for some edge cases. If your app requires Gson to do something special, head over to the Gson project[12] to look up the details.

## Customize Your Gson Configuration

Since this book is about Retrofit we won't go into the details of configuring Gson. Nonetheless, we want you to know that Gson has plenty of configuration options. Please go through the official User Guide[13], if you want to get an overview over all the options.

The good news is that Retrofit makes it very easy to customize Gson. You don't even need to implement your own custom converter, which we'll look at later in this chapter. In the previous section, we've explained that you can add the Gson converter like this:

```java
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
```

We didn't tell you, that you can pass a `Gson` instance to the `GsonConverterFactory.create()` method. Here's an example with various Gson configurations:

[12]https://github.com/google/gson
[13]https://github.com/google/gson/blob/master/UserGuide.md

```java
 Gson gson = new GsonBuilder()
     .registerTypeAdapter(Id.class, new IdTypeAdapter())
     .enableComplexMapKeySerialization()
     .serializeNulls()
     .setDateFormat(DateFormat.LONG)
     .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)
     .setPrettyPrinting()
     .setVersion(1.0)
     .create();

Retrofit retrofit = new Retrofit.Builder()
     .baseUrl("https://api.github.com")
     .addConverterFactory(GsonConverterFactory.create(gson))
     .build();
```

If you pass a `Gson` instance to the converter factory, Retrofit will respect all of your configurations. This makes it very easy to do the fine-tuning of your JSON mapping with Gson, while keeping the actual code changes to a minimum.

While JSON is currently a popular format, not all APIs use JSON. Another regularly used format is XML. In the next section, we'll go through the process of setting up an XML converter.

## Integrate an XML Converter

XML is a standard format for sending and receiving data to and from APIs. Retrofit makes it very easy to accept XML as a format. Similar to JSON, your first task is to add a Gradle dependency for the converter, because Retrofit doesn't ship with one build-in. Edit your `build.gradle` to add the XML converter:

```
dependencies {
    // Retrofit & OkHttp
    compile 'com.squareup.retrofit2:retrofit:2.0.0'

    // XML Converter
    compile 'com.squareup.retrofit2:converter-simplexml:2.0.0'
}
```

Simple XML[14] is a fast XML converter. After you've synced your project in Android Studio, the `SimpleXmlConverterFactory` class is available. Create an instance of it via the static `create()` method and pass it to Retrofit's builder method `addConverterFactory()`:

[14]http://simple.sourceforge.net/

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

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

If your app deals with multiple formats, you can call the `addConverterFactory()` multiple times with each format:

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com")
        .addConverterFactory(SimpleXmlConverterFactory.create())
        .addConverterFactory(GsonConverterFactory.create())
        .build();

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

With this setup, Retrofit would be able to deal with JSON and XML formats. There is no conflict, since these are distinct formats. However, if you add two JSON converters, Retrofit will go through the list and use the first available candidate. Thus, the order of `addConverterFactory()` calls matter! Retrofit will go through the converters in the same order you've added them. This can lead to an unexpected behavior, but we'll go into more detail on this topic later on.

In either case, Retrofit is now ready to parse the XML format. However, Simple XML requires you to annotate your Java objects to map the XML data. If your Java object classes aren't prepared yet, the next section will give you a short introduction.

## Simple XML Object Annotation

Retrofit will map your responses to Java objects; it doesn't matter which converter you're using. For XML responses, we need to annotate the Java objects for correct tag-to-property mapping.

We'll use an example `Task` class below and add annotations to map the corresponding `tasks.xml` file.

**tasks.xml**

This XML file doesn't appear to have any style information associated with it. The document tree is shown below.

```
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
    <task>
        <id link="http://url-to-link.com/task-id">1</id>
        <title> Retrofit XML Converter Blog Post</title>
        <description> Write blog post: XML Converter with Retrofit</description>
        <language>de-de</language>
        <link>http://www.ikn2020.de</link>
    </task>
</rss>
```

### Task

```
@Root(name = "task")
public class Task {
    @Element(name = "id")
    private long id;

    @Element(name = "title")
    private String title;

    @Element(name = "description")
    private String description;

    @Attribute(required = false)
    private String link;

    public Task() {}
}
```

The annotations within the `Task` class define the XML element name which gets mapped to the class property. Within the example, all elements from the Java class have the same name as in the XML file. Keep in mind that you can use different names and just put the correct XML element name to the `@Element` annotation.

This gives you a feeling on how to prepare your Java object classes to enable parsing with Simple XML. If your data is more complex, you might want to look at the official docs[15] to understand how to annotate your classes correctly.

Of course, there are more than just JSON and XML. The Retrofit creators provide a list of converters. We can't go through every single one of them, so we just give you an overview in the next section.

[15]http://simple.sourceforge.net/download/stream/doc/examples/examples.php

## Integrate Other Standard Formats

Besides JSON or XML, Retrofit can be configured with various converters and content formats. The retrofit-converters[16] directory lists all official response converter implementations:

**Gson**

Gson is for JSON mapping and can be added with the following dependency:

```
compile 'com.squareup.retrofit2:converter-gson:2.0.0'
```

**Simple XML**

Simple XML[17] is for XML mapping. You'll need the following line for your `build.gradle`:

```
compile 'com.squareup.retrofit2:converter-simplexml:2.0.0'
```

**Jackson**

Jackson[18] is an alternative to Gson and claims to be faster in mapping JSON data. The setup offers you a lot more customization and might be worth a look. You can add it with:

```
compile 'com.squareup.retrofit2:converter-jackson:2.0.0'
```

**Moshi**

Moshi[19] is another alternative to Gson. It's created by the developers of Retrofit. Moshi is based on Gson, but differentiates itself with some simplifications. If you want to give this young new player on the market a try, add it with:

```
compile 'com.squareup.retrofit2:converter-moshi:2.0.0'
```

**Protobuf**

Protobuf is build for the format Protocol Buffers[20], which is a new mechanism of serializing structured data. Protocol Buffers is a new generation and comes with a lot of upside. If your API already utilizes it, you can quickly add support on the app side with:

[16]https://github.com/square/retrofit/tree/master/retrofit-converters
[17]http://simple.sourceforge.net/
[18]https://github.com/FasterXML/jackson
[19]https://github.com/square/moshi/
[20]https://developers.google.com/protocol-buffers/

```
compile 'com.squareup.retrofit2:converter-protobuf:2.0.0'
```

#### Wire

Wire[21] is also for the Protocol Buffers format. It's developed by the creators of Retrofit and brings their vision of code to this converter. If you're interested, take a look by adding the dependency:

```
compile 'com.squareup.retrofit2:converter-wire:2.0.0'
```

#### Scalars

Finally, if you're overwhelmed by all these complex data structures and just want to map the basic Java primitives, you can use Scalars:

```
compile 'com.squareup.retrofit2:converter-scalars:2.0.0'
```

Retrofit is open and allows other converters. We'll look into developing our own custom converter in a second. Of course, you can provide your converter to the developer community and expand the list of available converters. For example, a third-party library is this converter[22], which adds support for the Logan Square[23] JSON serialization library. While Retrofit makes it very easy to support multiple data formats, there are a few things you need to keep in mind in order to preserve everything working. In the next section we'll give you an overview on how to work with multiple converters.

## Dealing with Multiple Converters

Retrofit accepts various structured data formats by outsourcing the conversion of those to independent converters. These converters usually only have one specific purpose. For example, Gson deals with the JSON format, while SimpleXML transforms XML data. Retrofit makes it easy by allowing the developer to call `addConverterFactory()` multiple times. As long as the converters are conflict free, e.g. Gson vs. SimpleXML, there won't be any issues. But what happens if you add two converters, which do more or less the same job, e.g. Moshi vs. Gson?

Retrofit has a simple way of dealing with this issue. First of all, it checks with every converter if it's able to deal with this specific data type. If the passed converter cannot understand some data, it won't be considered for the request (but will be asked again for the next request). The order of checks is done by first-come first-serve. That means the first converter you pass to Retrofit with `addConverterFactory()` will be checked first. If the first converter accepts the challenge, the rest of the list will not be asked.

[21]https://github.com/square/wire
[22]https://github.com/aurae/retrofit-logansquare
[23]https://github.com/bluelinelabs/LoganSquare

Effectively, this means that you need to be very careful when adding the converters! Make sure you specify the special-purpose converters with limited abilities first and general converters (like Gson) last.

You'll get a better feeling on how the details of Retrofit's design once we go through the next section: implementing our custom converter.

## Integrate Your Own Custom Converter

Do you still have an unfilled need for a data format after looking at all the provided converters? Then this section is for you! We'll go step-by-step through the process of implementing our own custom converter.

As you've seen in the previous section, there are a bunch of data structures and even more data structure converter libraries out there. They differentiate in a lot of factors and make a general description of this chapter difficult. In this section, we'll show you how an implementation for JSON with the Gson library would look like. Some details might look different for your data structure and your own implementation. We'll point out significant parts where it's possible that you need to deviate from our guide. Additionally, after reading this section, you can head over to the implementation of the official converters[24], which give you quite a bit of code examples.

You already know where our journey for the actual implementation starts:

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com")
        .addConverterFactory(new CustomConverter())
        .build();
```

The first step is to develop our `CustomConverter` class. Since we're not supporting any passed parameters, we can use the standard constructor. Of course, if you prefer, you can implement and instantiate it with `CustomConverter.create()`.

### Implementation of CustomConverter

The `CustomConverter` needs to extend Retrofit's `Converter.Factory`. The `Factory` interface lets us implement two important methods:

[24]https://github.com/square/retrofit/tree/master/retrofit-converters

```java
public class CustomConverter extends Converter.Factory {

    @Override
    public Converter<ResponseBody, ?> responseBodyConverter(
        Type type,
        Annotation[] annotations,
        Retrofit retrofit) {
            // todo: add code for our custom converter
            return super.responseBodyConverter(type, annotations, retrofit);
    }

    @Override
    public Converter<?, RequestBody> requestBodyConverter(
        Type type,
        Annotation[] parameterAnnotations,
        Annotation[] methodAnnotations,
        Retrofit retrofit) {
            // todo: add code for our custom converter
            return super.requestBodyConverter(
                type,
                parameterAnnotations,
                methodAnnotations,
                retrofit);
    }
}
```

The `requestBodyConverter` function is responsible for taking a Java object and returning the representative of that object in the chosen data format. For example, the method `requestBodyConverter` would take a `Task` object from the previous sections of this chapter, create the JSON representation and return that back to Retrofit. Retrofit can then use it to make the web service call.

The other function is the `responseBodyConverter` method. Its job is the opposite direction. It takes the structured data format, reads it and creates an appropriate Java object. Retrofit uses this method for parsing the web service responses in order to pass an easy-to-use Java object to the callback.

Of course, you've seen how convenient this is in the previous chapters and with your past experience with Retrofit. But behind all the magic is a lot of work.

## Limit the Converter Compatibility

Before we jump into the details, there is one more detail you should know: as we've mentioned before, Retrofit automatically goes through the list of passed configured converters and uses the first appropriate converter. It's important that your converter only accepts inputs, which is able to handle. Only if your converter limits itself to its true abilities, Retrofit can work so brilliantly with multiple data formats. This also means if your converter can only transform into one direction, you should only implement that direction and disable the other one.

## ResponseBodyConverter

Let’s go back to our `CustomConverter` class:

```java
public class CustomConverter extends Converter.Factory {

    @Override
    public Converter<ResponseBody, ?> responseBodyConverter(
        Type type,
        Annotation[] annotations,
        Retrofit retrofit) {
            // todo: add code for our custom converter
            return super.responseBodyConverter(type, annotations, retrofit);
    }

    @Override
    public Converter<?, RequestBody> requestBodyConverter(
        Type type,
        Annotation[] parameterAnnotations,
        Annotation[] methodAnnotations,
        Retrofit retrofit) {
            // todo: add code for our custom converter
            return super.requestBodyConverter(
                type,
                parameterAnnotations,
                methodAnnotations,
                retrofit);
    }
}
```

The first and probably more relevant direction and method is `responseBodyConverter`. It expects various parameters describing the incoming data (but not the data itself) and returns a `Converter` object which is able to transform the data structure into a Java object.

As promised, we’ll start on the implementation of the `responseBodyConverter`. It’s important to make sure you only return a `Converter<ResponseBody, T>` instance if your converter can handle it. Thus, we should check the `Type type` parameter. For example, if your converter can only handle regular `Class` objects, do a quick check if the incoming type is a `Class` type:

```java
if (!(type instanceof Class<?>)) {
    // if the custom converter cannot handle this type: return null
    // Retrofit will move on to the next converter
    return null;
}
```

Remember, when you implement a Retrofit Call object, it is typed in your expected Java class. Before your converter can actually do the conversion, Retrofit is nice and asks it, if it's able to do the conversion. Thus, the type check we're doing here is not based on the incoming JSON/XML/…, it's purely on the data type we want to convert into.

If you return `null`, Retrofit will understand it as sign that your converter is not compatible with this data type and move onto type-checking the next configured converter.

Depending on your data type, there is a lot of variety in the type-checks. Gson, which we'll use in this example, can deal with a lot of types, we don't need any further checks.

Once you're through with your checks and are sure your converter can handle the data, you need to prepare the actual conversion. Gson simply needs an instance and the description of the data type in Gson's `TypeAdapter` class:

```java
Gson gson = new Gson();
TypeAdapter<?> gsonAdapter = gson.getAdapter(TypeToken.get(type));
```

The final, and biggest hurdle is to return an instance of the typed `Converter` class. We'll need to create a new `CustomResponseBodyConverter<T>` class for it:

```java
private class CustomResponseBodyConverter<T>
        implements Converter<ResponseBody, T> {
    private TypeAdapter<T> gsonAdapter;
    private Gson gson;

    // constructor we pass the Gson instance and the type
    public CustomResponseBodyConverter(Gson gson, TypeAdapter<T> gsonAdapter) {
        this.gson = gson;
        this.gsonAdapter = gsonAdapter;
    }

    @Override
    public T convert(ResponseBody value) throws IOException {
        // the actual magic happens here
        // Gson takes the input and reads it as JSON
        JsonReader jsonReader = gson.newJsonReader(value.charStream());

        try {
            // Gson's read() method does all the work and returns a Java object
            // with other converters this part might be much more complex
            return gsonAdapter.read(jsonReader);
        } finally {
            value.close();
        }
    }
}
```

The converter only needs one method `convert(ResponseBody value)`. The `ResponseBody` is data representation of the underlying OkHttp framework. Basically, it contains the server response, which in our case is a JSON string.

The implementation of the `convert` method is 100% dependent on the data structure and the library you want to utilize. With Gson it's really easy, because it's just a few lines. With other libraries and data types, this might be more complex. You just need to make sure that it can handle the incoming `ResponseBody` and returns the Java object of the correct type.

Once you got the implementation of your `CustomResponseBodyConverter<T>` class, you can create an instance of it and you're all done with the `responseBodyConverter` part! If you want to see everything put together, take a look at the complete code snippet:

```
@Override
public Converter<ResponseBody, ?> responseBodyConverter(
        Type type,
        Annotation[] annotations,
        Retrofit retrofit) {
    // if your custom converter can only handle classes\
    // and not complex types, like List<Task>
    if (!(type instanceof Class<?>)) {
        // if the custom converter cannot handle this type: return null
        // Retrofit will move on to the next converter
        return null;
    }

    // depends on your implementation you might want to do further checks
    // ...

    // if all your checks go through
    // create the necessary fields for your data structure
    if (gson == null) {
        gson = new Gson();
    }

    TypeAdapter<?> gsonAdapter = gson.getAdapter(TypeToken.get(type));

    // finally, pass an instance of the response converter class
    return new CustomResponseBodyConverter<>(gson, gsonAdapter);
}
```

Unfortunately, in most use cases you'll also need to implement the other direction: from a Java object to the data structure representation, e.g. JSON. We'll look at the details in the next section.

## RequestBodyConverter

If you've worked through the details of the `responseBodyConverter`, the implementation of the `requestBodyConverter` will be no problem. You can re-use the majority of the code. You'll need to override the `requestBodyConverter` method. Just like before, Retrofit will provide the type and the Retrofit instance along with some other annotations for special use-cases.

Once again, you should check the type to make sure your converter can handle the incoming Java data and is able to create an appropriate request body for it.

After you've implemented the checks and initialized the data structure conversion library, you'll need to return an instance of the `Converter<T, RequestBody>` class:

```
@Override
public Converter<?, RequestBody> requestBodyConverter(
        Type type,
        Annotation[] parameterAnnotations,
        Annotation[] methodAnnotations,
        Retrofit retrofit) {
    // same type compatibility checks as the response converter

    // if your custom converter can only handle classes
    // and not complex types, like List<Task>
    if (!(type instanceof Class<?>)) {
        // if the custom converter cannot handle this type: return null
        // Retrofit will move on to the next converter
        return null;
    }

    // depends on your implementation you might want to do further checks
    // ...

    // if all your checks go through
    // create the necessary fields for your data structure
    if (gson == null) {
        gson = new Gson();
    }

    TypeAdapter<?> gsonAdapter = gson.getAdapter(TypeToken.get(type));

    // finally, pass an instance of the response converter class
    return new CustomRequestBodyConverter<>(gson, gsonAdapter);
}
```

As you've already recognized, the return is also an instance of the `Converter` class, just with a flipped type. Correspondingly, the `convert` method also has the same logic, just with flipped input and output types.

Just like the response conversion, the implementation of the `convert` method really depends on the data structure and your library. Since we're using Gson, this is really easy:

```
private class CustomRequestBodyConverter<T>
        implements Converter<T, RequestBody> {
    private final MediaType MEDIA_TYPE =
        MediaType.parse("application/json; charset=UTF-8");
    private final Charset UTF_8 = Charset.forName("UTF-8");

    private TypeAdapter<T> gsonAdapter;
    private Gson gson;

    public CustomRequestBodyConverter(Gson gson, TypeAdapter<T> gsonAdapter) {
        this.gson = gson;
        this.gsonAdapter = gsonAdapter;
    }

    @Override
    public RequestBody convert(T value) throws IOException {
        // create buffers
        Buffer buffer = new Buffer();
        Writer writer = new OutputStreamWriter(buffer.outputStream(), UTF_8);

        // take the incoming Java object and create a Json out of it
        // thanks to Gson this is easy
        JsonWriter jsonWriter = gson.newJsonWriter(writer);
        gsonAdapter.write(jsonWriter, value);
        jsonWriter.close();

        // finally, create an OkHttp RequestBody
        // pass the correct media type and the buffer
        return RequestBody.create(MEDIA_TYPE, buffer.readByteString());
    }
}
```

One line you might not have seen before is the last one. Because we need to return a `RequestBody` object, you'll need to provide your converted data structure in a compatible way. Besides the required HTTP MediaType header, OkHttp's `RequestBody` accepts a variety of input values via the static `create()` method. Compatible types for your data structure are `String`, `File`, `byte[]` and okio[25]'s `ByteString`.

## Usage of CustomConverter

After some hard work we want to see the benefits. You've worked through this section, so we want to reward you. After creating the three required classes adding your custom converter is just one more line:

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com")
        .addConverterFactory(new CustomConverter())
        .build();
```

Of course, you can add more converters before or after your custom one. Retrofit handles multiple converters very gracefully.

## Advanced Example of CustomConverter

Finally, let us give you a real-world example of implementing your own custom converter. We recently had the challenge that one of our APIs returned a list of containers. The container was a general class which had a few specializations. The issue was that the list the API returned had a mix of various types — the list was polymorph.

If we reduce it to a more visual example, let's assume we've three classes:

[25]https://github.com/square/okio

```java
public static class Animal {
    private String name;
    private String type;

    public String makeSound() {
        return "";
    }
}

public static class Dog extends Animal {
    private int amountOfCatToys;

    @Override
    public String makeSound() {
        return "Bark!";
    }
}

public static class Cat extends Animal {
    private boolean needsToGoOnAWalk;

    @Override
    public String makeSound() {
        return "Meow!";
    }
}
```

The API list returns a list of `Animals`, while some of them are from the type `Dog` and others are from the type `Cat`. We know which type it is based on the `Animal` class property `type`.

Our goal was that Retrofit and Gson will be configured in a way that they map all the containers into the appropriate class and also map all the type-specific properties. A simple version of the API JSON would look like this:

```json
[
  {
    "name": "cat 1",
    "type": "cat",
    "amountOfCatToys": 3
  },
  {
    "name": "cat 2",
    "type": "cat"
    "amountOfCatToys": 7
  },
  {
    "name": "dog 1",
    "type": "dog"
    "needsToGoOnAWalk": true
  }
]
```

As you can see, the cat objects have a different property than the dog object. Even though we're expecting a list with various types, we've to declare it with the super type `List<Animal>`. By default, Retrofit and Gson would only map the properties of the `Animal` class.

We've to adjust our custom converter from the previous section to make this work for us:

```java
public class PolymorphicCustomConverter extends Converter.Factory {
    private Gson gson;

    @Override
    public Converter<ResponseBody, ?> responseBodyConverter(
        Type type,
        Annotation[] annotations,
        Retrofit retrofit) {
        // depends on your implementation you might want to do further checks
        // ...

        // if all your checks go through
        // create the necessary fields for your data structure
        if (gson == null) {
            // setup a custom type adapter for our polymorph class
            final RuntimeTypeAdapterFactory<Animal> typeFactory =
                RuntimeTypeAdapterFactory
                    // Here you specify which is the parent class
                    // and what field particularizes the child class.
                    .of(Animal.class, "type")
                    // if the flag equals the class name, you can skip the
                    // second parameter. This is only necessary, when the
                    // "type" field does not equal the class name.
                    .registerSubtype(Dog.class, "dog")
                    .registerSubtype(Cat.class, "cat");

            // add the polymorphic specialization
            gson = new GsonBuilder()
```

```
                .registerTypeAdapterFactory(typeFactory).create();
        }

        TypeAdapter<?> gsonAdapter = gson.getAdapter(TypeToken.get(type));

        // finally, pass an instance of the response converter class
        return new CustomConverter.CustomResponseBodyConverter<>(gson, gsonAdapt\
er);
    }

    @Override
    public Converter<?, RequestBody> requestBodyConverter(Type type, Annotation[\
] parameterAnnotations, Annotation[] methodAnnotations, Retrofit retrofit) {
        // depends on your implementation you might want to do further checks
        // ...

        // if all your checks go through
        // create the necessary fields for your data structure
        if (gson == null) {
            gson = new Gson();
        }

        TypeAdapter<?> gsonAdapter = gson.getAdapter(TypeToken.get(type));

        // finally, pass an instance of the response converter class
        return new CustomConverter.CustomRequestBodyConverter<>(
            gson,
            gsonAdapter);
    }
}
```

This is a lot of code, but you're already familiar with 90% of it. The new section is the Gson setup, which now changed from:

```
Gson gson = new Gson();
```

to

```java
// setup a custom type adapter for our polymorph class
final RuntimeTypeAdapterFactory<Animal> typeFactory = RuntimeTypeAdapterFactory
    // Here you specify which is the parent class and
    // what field particularizes the child class.
    .of(Animal.class, "type")
    // if the flag equals the class name, you can skip the second parameter.
    // This is only necessary, when the "type" field does not
    // equal the class name.
    .registerSubtype(Dog.class, "dog")
    .registerSubtype(Cat.class, "cat");

// add the polymorphic specialization
Gson gson = new GsonBuilder().registerTypeAdapterFactory(typeFactory).create();
```

You might not understand what's going on here right away, but it basically declares that a list of `Animals` can contain sub types (of `Cat` and `Dog`) and that Gson should map them accordingly. Once we configured our new `PolymorphicCustomConverter` as a Retrofit converter, all of our API lists will come with the correct properties and in the right classes. Awesome!

What you need to adjust in your custom converter is totally up to your use case. There are plenty of scenarios in our mind, but that would go beyond the scope of this book. We hope this and the previous sections gave you the converter fundamentals to program the code for your own development work.

## Chapter Summary

In this chapter, you've learned how to Retrofit uses various converters to understand all kinds of data formats. You've learned how to quickly add and remove converters for your Retrofit implementation. We've shown you the usage of two standard converters in detail, one for JSON and one for XML. You've also seen a list of standard converters for other data formats and alternatives for JSON. Finally, we've implemented our own custom converter.

While in 90% of the cases you only need a few lines from this chapter and everything else works right away, in some edge cases you'll need to look deeper into the topic of converters. This chapter has shown you everything you need to know. Make sure you understood all the important concepts:

- [x] What are Retrofit converters
- [x] How to integrate standard Retrofit converters
- [x] How to integrate JSON & XML converters
- [x] How to customize the Gson converter
- [x] Why the order in which the converters are added matters
- [x] Get an overview of other available converters
- [x] Understand how to implement your own custom converter

In the next chapter, we get our hands on a common use case: user authentication. We’ll explain the authentication flow for basic and token authentication and additionally guide you through the usage of OAuth on Android. Finally, we’ll cover Hawk authentication.

# Chapter 4 — Authentication

You've already learned the essential basics of Retrofit and you're familiar with the important parts like request creation and execution and how to handle responses. Within this chapter, we're going to have a look at a very common use-case which is integrated in almost every Android app: authentication. We show you how to integrate four different types of authentication within your app. Starting out with **basic** authentication (username and password), we're moving to **token** authentication along with **OAuth 2** and finally touch the details of **Hawk** authentication on Android.

We've added an Android project within the extras of this book on Leanpub. You'll find the complete code including examples for each authentication type within this project. Feel free to use and adapt the code within your Android app.

Before we dive deep into the four authentication types, we need to create the foundations for this chapter first. We'll heavily make use of OkHttp's interceptors to integrate authentication. Therefore, we'll implement our own `AuthenticationInterceptor`, described below.

## Authentication Interceptor

The following sections explain how to add various authentication methods to your Android app. You can pick the desired type and adjust it to your needs. There's one thing that all authentication methods have in common: we intercept the request, add the individual credentials and move on with the actual request execution. And we'll leverage this consistency by creating a custom class that can be used for each authentication method.

Retrofit uses OkHttp for any network related operation and OkHttp enables the request interception with the help of customized `Interceptor` implementations. We'll use an `AuthenticationInterceptor` to add credentials in every request that requires authentication.

The following code snippets illustrates the request interception.

```
public class AuthenticationInterceptor implements Interceptor {

    private String authToken;

    public AuthenticationInterceptor(String token) {
        this.authToken = token;
    }

    @Override
    public Response intercept(Chain chain) throws IOException {
        Request original = chain.request();

        // set or override the `Authorization` header
        // keep the request body
        Request.Builder builder = original.newBuilder()
                .header("Authorization", authToken)
                .method(original.method(), original.body());

        Request request = builder.build();
        return chain.proceed(request);
    }
}
```

To make the `AuthenticationInterceptor` applicable as an OkHttp request interceptor, we need to implement the `Interceptor` interface and its `intercept` method. Within the `intercept` method, we have manipulate the request chain by adjusting the actual request to our needs. Instantiating an authentication interceptor requires at least a valid token value.

The actual interception consists of the fact that we create a new request based on the original one. We use the `Request.newBuilder()` method to clone the original request and add the `Authorization` header with a corresponding value. Remember that using the `.header()` method will override any pre-existing `Authorization` headers, because we want only one authentication token to be added with the request.

In the following, we'll make use of the `AuthenticationInterceptor` individually for each authentication method.

## Basic Authentication

You see basic authentication almost everywhere around the web and of course, this authentication method is used widely on mobile, too. Users sign up and log in to services with their username or email and password. It's fairly common to let users authenticate initially with their credentials and the backend sends an access token in return if authenticated successfully. We'll walk you through the usage of access tokens with Retrofit within the next section. For now, let's see how to integrate basic authentication.

Within the first chapter (Quick Start & Create a Sustainable REST Client), we've created an initial version of the Android REST client to perform API requests. We'll use the client foundation from the quick start chapter and enhance it with functionality for basic authentication.

If you didn't remember the sustainable REST client's code, don't worry. The code snippet below is a recap to remember the starting point for our `ServiceGenerator`.

```java
public class ServiceGenerator {

    private static final String BASE_URL = "https://api.github.com/"

    private static Retrofit.Builder builder =
            new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create());

    private static OkHttpClient.Builder httpClient =
            new OkHttpClient.Builder();

    public static <S> S createService(Class<S> serviceClass) {
        builder.client(httpClient.build());
        Retrofit retrofit = builder.build();
        return retrofit.create(serviceClass);
    }
}
```

You already know that Retrofit uses the `Retrofit` class and its integrated `Builder` to create customized API clients on Android. The previous snippet doesn't have support to send user credentials with the request. We'll extend the `ServiceGenerator` with this functionality within the following section.

## Integrate Basic Authentication

Due to the reason that we're adding support for basic authentication, we overload the `createService` method. We'll add two new parameters **username** and **password** besides the desired service class. Does your API uses a combination of email/password instead of username/password? No need to worry! Just adapt the code snippets if required and benefit from the depicted examples.

The basic approach of creating the client is the same as in the rudimentary approach: use Retrofit's builder as a helper to set the endpoint url, add Gson as a JSON response converter and create the OkHttp client for any HTTP requests and response handling.

The difference now: we add an OkHttp request interceptor to set an `Authorization` header value for the HTTP request. Nonetheless, we'll only add the credentials if username and password are existent. If you don't provide username and password, the service client will be created as we would do with the first method.

Here's the code snippet that extends the `ServiceGenerator` to integrate with basic authentication using the previously introduced `AuthenticationInterceptor`.

```java
public class ServiceGenerator {

    private static final String BASE_URL = "https://yourbaseurl.com/"

    private static Retrofit.Builder builder =
            new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create());

    private static OkHttpClient.Builder httpClient =
            new OkHttpClient.Builder();

    public static <S> S createService(Class<S> serviceClass) {
        return createService(serviceClass, null, null);
    }

    public static <S> S createService(
            Class<S> serviceClass, String username, String password) {

        if (!TextUtils.isEmpty(username) && !TextUtils.isEmpty(password)) {
            // use OkHttp's "Credentials" factory to create the basic auth token
            String credentials = Credentials.basic(username, password);

            // add auth interceptor using the created credentials
            httpClient.addInterceptor(
                new AuthenticationInterceptor(credentials));
        }

        builder.client(httpClient.build());
        retrofit = builder.build();

        return retrofit.create(serviceClass);
    }
}
```

The code looks sort of blown up related to the previous snippet, but if you scan through the content, you'll recognize that we just made the required adjustments to get things work. Let's walk through the changes: we needed to separate the OkHttp client from Retrofit's builder, because it's only allowed to add interceptors to OkHttp using its mutable builder instance. Further, we've added the overloaded `createService()` method accepting three parameters (service class, username, password). Within the new `createService()` method, we just check if the provided username and password aren't empty.

For the authentication part, we have to adjust the format of given username and password. Basic authentication requires both values as a concatenated string separated by a colon. Additionally, the newly created, concatenated string has to be Base64 encoded. If the variables contain actual values, we create the encoded credentials for basic authentication leveraging OkHttp's `Credentials` factory class. Afterwards, we create and add an instance of the `AuthenticationInterceptor` using the created credentials string.

Almost every web service and API evaluates the **Authorization** header of the HTTP request. That's why we set the encoded credentials value to that header field. If the API you're going to call with this client requires another header field to expect the user's credentials, adjust the header field to your needs within the `AuthenticationInterceptor` or just create your own interceptor implementation.

## Usage

To make things approachable, let's assume you defined a `LoginService` like shown in the code below.

```
public interface LoginService {
    @POST("login")
    Call<User> basicLogin();
}
```

The `LoginService` interface has only one method: `basicLogin`. It has the `User` class as the return value and doesn't expect any additional parameters.

Now you can create your HTTP client by passing your given credentials (username, password) to the new `createService` method within the `ServiceGenerator`.

```
String username = "myuser";
String password = "secretpassword";

LoginService loginService =
        ServiceGenerator.createService(LoginService.class, username, password);
Call<User> call = loginService.basicLogin();

call.enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        // check if response is successful
        // if yes: use the user object
        // if no: show error message
    }

    @Override
    public void onFailure(Call<User> call, Throwable t) {}
});
```

Of course, you'll use the provided username and password from an EditText or any other input source :) The `ServiceGenerator` method will create the HTTP client including an authentication interceptor. Using the call instance and executing the request, the provided credentials for basic authentication will be automatically passed to your defined API endpoint.

The actual request will be intercepted by OkHttp and your `AuthenticationInterceptor` adds the `Authorization` header to authenticate the user use with given credentials.

Basic authentication is mostly just the entry level to request user specific data and a unique authentication token. The following section shows you how to use this token to authenticate your requests.

## Token Authentication

Initially, most apps use username and password as the authentication method of choice and afterwards return an authentication token for future requests. Thus, we definitely need to look at token authentication. Using this authentication method, we avoid the risk of saving users private credentials (including the password) on client-side. We just use the access token to request data from API endpoints.

**Remember**: storing confidential information on client-side requires always a lot attention. As an app developer, you don't know where users are connecting with the Internet and with that if their secret access token leaks. The user-specific access token is mostly the only "security check" before having entrance to all the users data. You need to minimize the amount of privileged data on the user's device and maximize the effort to save the credentials as secure as possible!

### Integrate Token Authentication

If you read the previous section about basic authentication with Retrofit, you'll probably guess what we're going to do: extend the `ServiceGenerator` class and integrate a method that handles token authentication appropriately. Let's overload the `createService()` method again that accepts a string value representing the authentication token.

```java
public class ServiceGenerator {

    private static final String BASE_URL = "https://yourbaseurl.com/"

    private static Retrofit.Builder builder =
            new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create());

    private static OkHttpClient.Builder httpClient =
            new OkHttpClient.Builder();

    public static <S> S createService(Class<S> serviceClass) {
        return createService(serviceClass, "");
    }

    public static <S> S createService(
            Class<S> serviceClass, String username, String password) {

        String credentials = null;

        if (!TextUtils.isEmpty(username) && !TextUtils.isEmpty(password)) {
            // use OkHttp's "Credentials" factory to create the basic auth token
            credentials = Credentials.basic(username, password);
        }

        return createService(serviceClass, credentials);
    }

    public static <S> S createService(
            Class<S> serviceClass, String authToken) {

        if (!TextUtils.isEmpty(authToken)) {
            // add auth interceptor using the created credentials
            httpClient.addInterceptor(new AuthenticationInterceptor(authToken));
        }

        builder.client(httpClient.build());
        retrofit = builder.build();

        return retrofit.create(serviceClass);
    }
}
```

The updated `ServiceGenerator` looks even more complex than it actually is. We did some minor refactoring to the existing code and added a new `createService(serviceClass, authToken)` method. To minimize duplicated code snippets, we've reused the existing code block of building the final OkHttpClient and Retrofit instances from their Builder objects. Ultimately, the `createService` method returns the created instance of a given service class.

As you can see, we pass the token value as a `String` variable into the new `createService(serviceClass, authToken)` method. Afterwards, we check if the `authToken` parameter contains an actual value. If yes, we add a new instance of the `AuthenticationInterceptor` with the current `authToken` value to the list of OkHttp's request interceptors. If you can't remember the content of the `AuthenticationInterceptor`, please head over to the beginning of this chapter to recap the code.

The authentication interceptor set's the **Authorization** header with the given `authToken` value for each request. If you're using another HTTP header field or a query parameter for your authentication token, either adjust the authentication interceptor's code or create a new one that does the desired job.

Now, every HTTP service client created with a `createService()` method including an accurate authentication token (that also includes a username and password combination) will automatically pass the token value to your API endpoints with any request.

## Example Usage

Let's assume we have an API that has an endpoint located at `/me` and returns scoped data for the user that requests data with a valid authentication token. On Android, the `UserService` interface will look like the following code snippet.

```
public interface UserService {
    @GET("me")
    Call<User> me();
}
```

The API you're going to call awaits requests at the endpoint `https://api.example.com/me` and requires authentication to get user data in response. Now, let's create a user service object and do the actual request.

```java
String token = "your-secret-auth-token";

UserService userService =
        ServiceGenerator.create(UserService.class, token);

Call<User> call = userService.me();
call.enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        // check if response is successful
        // if yes: use the data about yourself :)
    }

    @Override
    public void onFailure(Call<User> call, Throwable t) {}
});
```

The code snippet above generally illustrates how to use the `ServiceGenerator` to create your service client with a given authentication token. Of course, you have to replace some variables to make the example work within your app.

As you can see, authenticated requests aren't that complex on Android. The request interceptor appends an authentication token if required. If you read both sections on basic and token authentication, you've seen that it's actually just a matter of how the API wants to receive the authentication information. You need to adjust your client code appropriately and Retrofit will do the work with the help of OkHttp.

## OAuth 2

This section describes and illustrates how to authenticate against an OAuth API from your Android app. You know many services that use OAuth for authentication when interacting with their platforms. A well known platform is Twitter: they have many third-party clients which use OAuth and require you to allow the app access to your data.

We won't go into much detail about OAuth[26] itself. Please follow the linked website for more information about OAuth and its details. Nonetheless, we need a basic understanding and describe the principles and necessary details to understand the authentication flow.

**Hint**: most platforms provide an Android SDK for development against their APIs. Mostly, there's already an authentication mechanism integrated which simplifies your setup on client-side. If you still want or need to develop your own OAuth client for a specific API, please read on :)

[26]http://oauth.net/

## OAuth 2 Basics

OAuth is a token based authentication method which uses an access token for interaction between the user and API. OAuth requires several steps and requests against the API to get your access token.

**OAuth Steps**

1. Register an app for the API you want to develop. Use the developer sites of the public API you're going to develop for
2. Save the client id and the client secret in your app
3. Request access to user data from your app
4. Use the authorization code to get the access token
5. Use the access token for future API interactions

## Register Your App

Before starting with the implementation of an OAuth client on Android, you have to register your app for the service or API you want to develop. Once the sign up for your application (which you're going to build) is finished, you'll receive a **client id** and **client secret**. Both values are required to authenticate your app against the selected API. Save both values (client id and secret) within your app, like a string constant within your Java code.

Please don't use the `strings.xml` resources to store both credentials, because that's a huge security gap. Use a Java constant to save the values and enable ProGuard to obfuscate your code to decrease the likelihood of leaking both values.

## Create Your Project

We'll assume you already have an existing Android project. If you don't, just go ahead and create a new one. When you're done, move on to the next section and get ready for coding :)

## Integrate OAuth

Since we're using the `ServiceGenerator` class from the basic authentication section, we'll further extend it and add a method to handle the OAuth access token. Again, we need to extend the `ServiceGenerator` with another method that accepts the `AccessToken` as a parameter. The snippet below shows the updated `ServiceGenerator` class.

We'll simplify the `ServiceGenerator` representation to reduce the amount of shown code. That doesn't mean you should delete the previously created methods for basic or token authentication, because you'll need especially the basic authentication for OAuth as well.

```java
public class ServiceGenerator {

    private static final String BASE_URL = "https://yourbaseurl.com/"

    private static Retrofit.Builder builder =
            new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create());

    private static OkHttpClient.Builder httpClient =
            new OkHttpClient.Builder();

    public static <S> S createService(
            Class<S> serviceClass, String username, String password) {
        // create basic auth credentials
        return createService(serviceClass, credentials);
    }

    public static <S> S createService(
            Class<S> serviceClass, AccessToken token) {

        String authToken = null;
        if (token != null) {
            authToken = token.getTokenType().concat(token.getAccessToken());
        }
        return createService(serviceClass, authToken);
    }

    public static <S> S createService(
            Class<S> serviceClass, String authToken) {

        if (!TextUtils.isEmpty(authToken)) {
            // add auth interceptor using the created credentials
            httpClient.addInterceptor(new AuthenticationInterceptor(authToken));
        }

        builder.client(httpClient.build());
        retrofit = builder.build();
        return retrofit.create(serviceClass);
    }
}
```

Again, we're using the `AuthenticationInterceptor` to set the **Authorization** header field. Using the `AccessToken`, the actual header field value consists of two parts: first, the token type which is **Bearer** for OAuth requests and second, the actual access token value.

As you can see in the code snippet above, the method requires an `AccessToken` as a second parameter. This class looks like this:

```java
public class AccessToken {
    private String accessToken;
    private String tokenType;

    public String getAccessToken() { return accessToken; }

    public String getTokenType() {
        // OAuth requires the first letter of Authorization HTTP
        // header value for token type to be uppercase
        if ( ! Character.isUpperCase(tokenType.charAt(0))) {
            tokenType = Character.toString(tokenType.charAt(0))
                    .toUpperCase() + tokenType.substring(1);
        }

        // return token type with trailing space
        // to separate token type and value
        return tokenType + " ";
    }
}
```

The `AccessToken` class consists of two fields: `accessToken` and `tokenType`. Because OAuth API implementations require the token type to be in uppercase, we check the styling first. If it doesn't fit, we update the token type style and let the token type begin with a capital letter. For example, your API returns **bearer** as the token type: any request with this token type style would result in either `401 Unauthorized`, `403 Forbidden` or `400 Bad Request` (if your API is developed following the specs, else it's another status code). That's why we need to check and adjust the token type in case the first letter isn't uppercase.

The HTTP header field will look like the following example when set correctly:

```
Authorization: Bearer <your-bearer-token>
```

## Integrate OAuth in Your App

First, we'll create a new activity called `LoginActivity`. You can use a simple view with only one button (layout code below). Here's the code for the new activity:

```java
public class LoginActivity extends Activity {

    // you should either define client id and secret
    // as constants or in string resources
    private final String clientId = "your-client-id";
    private final String clientSecret = "your-client-secret";
    private final String redirectUri = "your://redirecturi";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);

        Button loginButton (Button) findViewById(R.id.loginbutton);
        loginButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(
                        Intent.ACTION_VIEW,
                        Uri.parse(
                                ServiceGenerator.BASE_URL + "/login" +
                                        "?client_id=" + clientId +
                                        "&redirect_uri=" + redirectUri));

                startActivity(intent);
            }
        });
    }
}
```

You have to adjust the values for the class properties **clientId, clientSecret, redirectUri**. We'll get in more detail about the redirect URI in a second.

We set an `OnClickListener` for the login button within the `onCreate` method. Once the `onClick` event gets triggered, it creates a new intent showing a web view with the content defined at the given URI.

**Important Note**: you have to provide your client id and client secret in this request. Look at the API documentation how to pass the values with your request. The example above sends them as query parameters, because the API requires both for further processing and to associate your app with the request.

Additionally, check the `Uri.parse(…)` part. You have to point the url to the login (or authorize) endpoint to show the correct login or access rights screen.

The layout for `activity_login.xml` can be as simple as this.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Login"
        android:id="@+id/loginbutton"
        android:gravity="center_vertical|center_horizontal"
        />
</RelativeLayout>
```

It's just a single button on your view :)

### Define Activity and Intent Filter in AndroidManifest.xml

An intent in Android is a messaging object used to request action or information (communication) from another app or component. The intent filter is used to catch a message from an intent, identified by an intent's action, category and data.

The intent filter is required to make Android return to your app, so you can grab data from the response within your intent. That means, when starting the intent after clicking on your login button within your `LoginActivity`, this filter catches any response and makes additional information available. The code below shows the activity definition in `AndroidManifest.xml` including the intent filter for this activity.

```
<activity
    android:name="com.futurestudio.oauthexample.LoginActivity"
    android:label="@string/app_name"
    android:configChanges="keyboard|orientation|screenSize">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data
            android:host="redirecturi"
            android:scheme="your" />
    </intent-filter>
</activity>
```

We assume you're familiar with the definitions inside your `AndroidManifest.xml` file. Let's just glance over the snippet above and clarify the `intent-filter` part.

First, we define the action which is simply an android view (in our case a web view). The category definition allows interactions within the intent and makes the web view browsable to insert data within the upcoming login screen. The data definition has two android attributes: `host` and `scheme`. Those two attributes describe the schema of your redirect uri which you need to define while creating your app at the web service or API you're developing the app for.

**Remember**: one of the first steps within this section was to create your app within the developer sites of your API. When creating your new app, you need to define a redirect URI. Make sure you provided the same information as you use within your Android app (e.g. `your://redirecturi`).

The redirect url gets called once the login procedure finishes successfully and includes the authorization code. The authorization code is required to finally request the access token.

## Catch the Authorization Code

Ok, by now we have defined the intent to show the web view which presents as a **deny** or **allow** view once the user logged in with their personal credentials. You'll notice the style of this view when seeing it :)

Now we want to get the access token for further API interactions. This token is another two API requests away. First, we need to parse and use the returned authorization code which is part of the response when pressing the **allow** button within the intent's web view. The following method belongs to your `LoginActivity`. We separate the code snippet from the previously shown code, because it's easier to explain the contents and you don't get overloaded with duplicated code.

```java
@Override
protected void onResume() {
    super.onResume();

    // the intent filter defined in AndroidManifest
    // will handle the return from ACTION_VIEW intent
    if (getIntent() != null) {
        Uri uri = getIntent().getData();
        if (uri != null && uri.toString().startsWith(redirectUri)) {
            // use the parameter your API exposes for the code
            // (mostly it's "code")
            String code = uri.getQueryParameter("code");
            if (code != null) {
                // get access token
                // we'll do that in a minute
            } else if (uri.getQueryParameter("error") != null) {
                // show an error message here
            }
        }
    }
}
```

Your app returns into the `onResume` method of Android's lifecycle passing through the login and access request. Here you can see we're using the `getIntent().getData()` method to retrieve the intent's response.

Now, we don't want to run into any `NullPointerException` and check the values. Afterwards, we extract the authorization code from the response's query parameters.

Imagine the response url when clicking **allow** like

```
your://redirecturi?code=1234
```

and for **deny** access like

```
your://redirecturi?error=message
```

## Get Your Access Token

You're almost done, the access token is just one request away. Now that we have the authorization code, we need to request the access token by passing **client id**, **client secret** and **authorization code** to the API.

In the following, we extend the previous presented `onResume` method to do another API request. But first, we have to extend the `LoginService` interface and define a method to request the access token. We'll just extend the `LoginService` with another method called `getAccessToken`.

If you read the section on basic authentication, you're already familiar with the first method, `basicLogin()`. The second method `getAccessToken()` expects two parameters: the authorization code and a grant type.

```
public interface LoginService {
    @POST("login")
    Call<User> basicLogin();

    @POST("token")
    Call<AccessToken> getAccessToken(@Query("code") String authCode,
                                     @Query("grant_type") String grantType);
}
```

This is the interface definition for the `LoginService`. We'll use it to create our service client with the help of the previously defined `ServiceGenerator`. The `getAccessToken()` method expects two query parameters: `code` and a `grantType`. The authorization code is the one we got from our first request and the grant type is a single string value, like `authorization_code`.

Actually, there are 4 grant types for OAuth APIs:

1. **authorization code** for apps running on web servers
2. **implicit** for mobile or browser based apps
3. **password** for authentication with username and password
4. **client credentials** for app access

However, have a look at the API documentation of the service you're developing the app for and you'll find the information which grant type to use.

Finally, let's have a look at the complete code for the `onResume` method to request and receive the access token.

```java
@Override
protected void onResume() {
    super.onResume();

    // the intent filter defined in AndroidManifest
    // will handle the return from ACTION_VIEW intent
    if (getIntent() != null) {
        Uri uri = getIntent().getData();
        if (uri != null && uri.toString().startsWith(redirectUri)) {
            // use the parameter your API exposes for the code
            // (mostly it's "code")
            String code = uri.getQueryParameter("code");
            if (code != null) {
                // get access token
                LoginService loginService =
                        ServiceGenerator.createService(
                                LoginService.class, clientId, clientSecret);

                Call<AccessToken> call =
                        loginService.getAccessToken(code, "authorization_code");
                call.enqueue(new Callback<AccessToken>() {
                    @Override
                    public void onResponse(
                            Call<AccessToken> call,
                            Response<AccessToken> response) {
```

```
                        // save access token including token type
                        // start next activity via intent or do whatever you wan\
t :)
                    }

                    @Override
                    public void onFailure(Call<AccessToken> call, Throwable t) {}
                });
            } else if (uri.getQueryParameter("error") != null) {
                // handle error here
            }
        }
    }
}
```

If everything went fine, you have authenticated the user successfully and received the access token. Use this access token for further requests against the API. Use the added method within the `ServiceGenerator` to authenticate your requests appropriately by adding the correct `Authorization` header value.

Some APIs limit the lifetime of an access token. You'll receive the expiry information of the token within the response. Some API providers send an `expires_in` field with the number of seconds until the access token expires.

The next section describes how to handle expired tokens with the help of a provided refresh token.

## OAuth — Refresh Your Access Token

When working with OAuth APIs, you have to manage access tokens and sometimes also refresh tokens. Generally, the access token is what you'll use to authenticate your requests. However, the access tokens are short lived and the lifetime differs per API provider. Due to security reasons, the access token validity ranges from one hour to infinite (= if there's no refresh token handling required).

In contrast, the request token has a much longer lifetime. Depending on the API implementation, it might be infinite and you have to revoke access manually to render the access token useless.

The idea behind the two tokens for OAuth APIs has a single reason: security. If your access token got compromised, the attacker has a limited time window to (probably) misuse it. If the attacker has access to the refresh token besides access token, the situation is a lot more serious. If attackers have access to the refresh token only, there's less to worry, because the attacker needs the client id and client secret to request a new access token.

## Recognize Invalid Access Token

Besides security concerns, as a developer you have to recognize when the access token is expired. Using an out of date access token will result in a failing request. The response status code depends on the API implementation and will be either `400`, `401` or `403` (or `404` like GitHub does to prevent leakage of information about private repositories).

It's common practice to use the `401 Unauthorized` status code to indicate wrong credentials or expired access tokens.

## Refresh Invalid Access Token

Failing requests using an outdated access token result in a response which probably includes an error description. You need to check the developer documentation of your API provider how they handle expired access tokens and what comes in return. Most of the times, you'll receive an error object providing more information about the issue.

If your error response contains an error type like `{"errorType": "expired_token"}` or a message like `{"message": "Access token expired"}` that indicates the actual failure reason, you can use string comparisons to kickoff the correct handling.

Actually, you need to have a procedure in place to refresh the invalid access token in background and only show errors to the user if something goes wrong badly. The following sections will show you a bad approach and additionally our proposed solution.

### Bad Solution: Check if Response Code Is 401

Let's assume somewhere in your your app you start a request and authentication is required to successfully receive data. You pass the access token to the `ServiceGenerator` and the token value gets added to the request using the previously shown `AuthenticationInterceptor`. You didn't start the app a while and the latest access token is expired. Starting your app and requesting the data will result in an error and you have to interpret and handle this error appropriately.

Let's assume the following snippet is used within the source code. Further, we also assume that the API returns the status code `401` using an expired access token.

```
// read your token from shared preferences or other storage
AccessToken token = getMyAccessToken();

UserService userService =
        ServiceGenerator.createService(UserService.class, token);

Call<User> call = userService.me();
call.enqueue(new Callback<User>() {
    @Override
        public void onResponse(Call<User> call, Response<User> response) {
        // response unsuccessful with `401 Unauthorized`?
        if (! response.isSuccessful()) {
            if (response.code() == 401) {
                refreshAccessToken();
            }
        }
    }

    @Override
    public void onFailure(Call<User> call, Throwable t) {}
});
```

The shown code above might come to your mind at first when thinking about how to notice outdated access tokens. But that's just half of the solution and requires you to check for authentication errors within every callback. That means, each request in your app is followed by the same logic within the callback that checks if the response is successful and if not, if the status code is `401`. You immediately understand this costly approach and we need to find a place to check the response status once so the refresh token handling starts off automatically.

It would be very poor of us if we leave you with a bad solution and don't show you our approach on how to refresh expired access tokens. An elegant solution to approach the refresh token handling is OkHttp's authenticator.

## Good Solution: OkHttp Authenticator

As you've seen within the bad solution, checking for a successful response within every callback is an expensive operation that blows up your code base. We want a solution where the code to handle the refresh operation is written once and jumps in automatically if we receive an response with status code `401`.

An elegant solution is OkHttp's authenticator. It gets called if you receive a `401` response. That's exactly the behavior we want in our app: one location to refresh them all!

Actually, the solution to our problem consists of four steps:

1. Use OkHttp Authenticator (is called for `401` responses)
2. Request new access token applying the refresh token
3. Add authentication interceptor using the new access token
4. Repeat failed request with the new access token

Our custom OkHttp authenticator implementation is called `RefreshTokenAuthenticator` and needs to implement OkHttp's `Authenticator` interface. There's one method that we need to override, `authenticate()`. That's the place where all the magic happens: request the new access token and add a new authentication interceptor using the new access token and finally repeat the failed request with an updated `Authorization` header value that will succeed :)

### RefreshTokenAuthenticator

The constructor for `RefreshTokenAuthenticator` expects two parameters: the OkHttp builder to add a new interceptor and the refresh token value.

Within the `authenticate()` method, we first check if we've already failed at least two times with our request and in case we did: give up by returning `null`. Else, we execute a synchronous request using the refresh token to request a new access token. Afterwards, we add the new authentication interceptor for any further requests and additionally, we update the header value for the current request to use the new token and return this request for repeated execution.

```java
public class RefreshTokenAuthenticator implements Authenticator {

    private String refreshToken;
    private OkHttpClient.Builder httpBuilder;

    public RefreshTokenAuthenticator(
            OkHttpClient.Builder httpBuilder, String refreshToken) {
        this.httpBuilder = httpBuilder;
        this.refreshToken = refreshToken;
    }

    @Override
    public Request authenticate(Route route, Response response)
            throws IOException {
        // If we've failed 2 times, give up.
        if (responseCount(response) >= 2) {
            return null;
        }

        Request.Builder builder = response.request().newBuilder();

        // execute synchronous call to request a new access token
        UserService service = ServiceGenerator.createService(UserService.class);
        Call<AccessToken> call =
                service.refreshToken(refreshToken, "refresh_token");
        final AccessToken accessToken = call.execute().body();

        if (accessToken != null) {
            // safe new access token (shared pref, account manager, etc)

            // add auth interceptor using the new access token
            String authToken =
                    accessToken.getTokenType().concat(
                            accessToken.getAccessToken());
            httpBuilder.addInterceptor(
                new AuthenticationInterceptor(authToken));

            // repeat request with updated access token
            builder.header("Authorization", authToken);
        }

        return builder.build();
    }

    private int responseCount(Response response) {
        int result = 1;

        while ((response = response.priorResponse()) != null) {
            result++;
        }

        return result;
    }
}
```

The code above looks complicated, but it's actually just executing the four steps that we've outlined within the solution overview.

Keep in mind, that an occurring `IOException` from the synchronous request will return to your original `onFailure` callback. If the synchronous request is successful and you're receiving the new access token, make sure to store it within your app like shared preferences or Android's account manager.

**Attention**: storing and reading data from Android's shared preferences from multiple threads isn't safe. Shared preferences might return different values for different threads! Even the `MODE_MULTI_PROCESS` isn't 100% safe. If you want to use shared preferences, you need to make sure that everything works as intended with your thread setup. If not, you need to use a different thread-safe data storage.

That's it :) You've successfully implemented a custom OkHttp authenticator that handles the refresh for your access token in one place and gets called automatically for `401 Unauthorized` responses.

Interacting with OAuth APIs can cause pain due to expired access tokens and the subsequent refresh operation to request a valid one. Retrofit's callbacks are a bad place to apply this procedure, because it can occur throughout your app and you don't want to add the same error logic within every callback.

A much better solution is to benefit from OkHttp's authenticator which gets called for unsuccessful responses with status code `401`. That's exactly the place to put the logic to handle the refresh operation once.

## Hawk Authentication

Hawk authentication[27] is sort of the underdog in the area of authentication types. It's been officially around since November 2012 when Eran Hammer (aka hueniverse)[28] published the first version. Since than, the HTTP authentication scheme has evolved to its current version `3.x`.

### Hawk Authentication Introduction

Hawk is an HTTP authentication scheme allowing authenticated client requests with partial cryptographic request (and response) verification. The verification includes the HTTP method, request URI, host and optionally the request payload.

We won't dive deeper into the details of Hawk and assume that you're already familiar with the authentication scheme. And due to the fact that you're reading this part of the book, you've decided to give it a shot :) If you need more information about Hawk, please visit the GitHub project repository[29] and follow the information provided within the Readme.

### Add Hawk Dependencies

There are multiple implementation available for Node.js, Java, Ruby, Python and F#/.Net. Of course, we're using the Java implementation[30] on client-side to handle the hawk credentials and header generation on Android. Add the following line to your `build.gradle` and sync the project in Android Studio.

[27]https://github.com/hueniverse/hawk
[28]http://hueniverse.com/
[29]https://github.com/hueniverse/hawk
[30]https://github.com/wealdtech/hawk

```
compile 'com.wealdtech.hawk:hawk-core:1.2.0'
```

Once the synchronization process finishes, move on with the next paragraph and integrate Hawk into your project.

## Integrate Hawk Authentication

Generally, the implementation of Hawk authentication on Android is similar to every other authentication type (like basic, token, or OAuth). Depending on the implementation of your API, you'll need to send an appropriate value for the HTTP `Authorization` header field.

If you read the quick start chapter or previous sections within this chapter on authentication types, you're already familiar with our concept of the `ServiceGenerator`. We're using this class to generate service clients from your interface definitions.

Yet, we've created an overloaded `createService()` method for each authentication type. To keep focus on Hawk, we'll just skip the previous methods and only show the `createService()` method which does all the magic in regard to Hawk's authorization header generation.

```
public class ServiceGenerator {

    private static final String BASE_URL = "https://yourbaseurl.com/"

    private static Retrofit.Builder builder =
            new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create());

    private static OkHttpClient.Builder httpClient =
            new OkHttpClient.Builder();

    public static <S> S createService(
            Class<S> serviceClass, final HawkCredentials credentials) {

        if (credentials != null) {
            HawkAuthenticationInterceptor interceptor =
                    new HawkAuthenticationInterceptor(credentials);

            httpClient.addInterceptor(interceptor);
        }

        builder.client(httpClient.build());
        retrofit = builder.build();
        return retrofit.create(serviceClass);
    }
}
```

Let's review the code and what's actually going on here. The method parameters for the new `createService()` method are the service interface you want to be generated and second the Hawk credentials. As you can see, we use another OkHttp interceptor, specifically the `HawkAuthenticationInterceptor`, and pass it to the OkHttpClient instance.

The detailed code for the `HawkAuthenticationInterceptor` is shown in the following snippet:

```
public class HawkAuthenticationInterceptor implements Interceptor {

    private HawkClient hawkClient;

    public HawkAuthenticationInterceptor(HawkCredentials credentials) {
        hawkClient = new HawkClient.Builder()
                .credentials(credentials)
                .build();
    }

    @Override
    public Response intercept(Chain chain) throws IOException {
        Request original = chain.request();
        URI uri = original.url().uri();
        String method = original.method();

        // generate Hawk auth header
        String header =
                hawkClient.generateAuthorizationHeader(
                        uri, method, null, null, null, null);

        // add or override `Authorization` header value
        Request.Builder requestBuilder = original.newBuilder()
                .header("Authorization", header)  // <-- important line
                .method(method, original.body());

        Request request = requestBuilder.build();
        return chain.proceed(request);

    }
}
```

The required `HawkCredentials` are passed to the constructor and directly used to create an instance of the `HawkClient`. The `HawkClient` is responsible for the creation of the actual authorization value that will be sent with the request.

For the generation of the `Authorization` header, we need at least the request URI and request method. The used Java library will automatically add the required host information while generating the header.

Once the required authorization header is created, we're taking the original request as the base reference to create a new one. We set the appropriate header field and proceed the request chain using the original body.

## Usage

It's important to understand, that you need to request your Hawk credentials first before you're able to use the authentication method. Request your credentials as you would do with any other authentication type like using basic authentication via username and password. Your API should return the Hawk credentials in response which are required to create the `HawkClient` instance that generates the authorization value for further authenticated requests against API endpoints.

Let's look at the following interface definition which only describes the `me` endpoint requiring authentication before returning the `User` object.

```
public interface UserService {
    @POST("me")
    Call<User> me();
}
```

In the following, we assume that you've already requested your Hawk credentials from the API you're consuming. If you need to get those credentials first, you may need to start over this chapter and read more about basic authentication.

For illustration purposes, we use the test credentials from Hawk's GitHub repository. On server side, a basic Node.js server is doing the work and requires (Hawk) authentication for the `/me` endpoint.

```
HawkCredentials hawkCredentials =
    new HawkCredentials.Builder()
        .keyId("dh37fgj492je")
        .key("werxhqb98rpaxn39848xrunpaw3489ruxnpa98w4rxn")
        .algorithm(HawkCredentials.Algorithm.SHA256)
        .build();

UserService userService =
    ServiceGenerator.create(UserService.class, hawkCredentials);
```

```java

Call<User> call = userService.me();
call.enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        // check if response is successful
        // if yes: use the data about yourself :)
    }

    @Override
    public void onFailure(Call<User> call, Throwable t) {}
});
```

Use the previously mentioned Java library to create a `HawkCredentials` object. Pass this object to the `createService()` method while generating the service client using the `ServiceGenerator`. Get the `Call` object for your request and execute it (synchronous or asynchronous) to receive the response object. The defined OkHttp interceptor will add the authorization token before executing requests.

That's all the magic! You're now able to authenticate your requests using the Hawk scheme.

We've already covered basic and token authentication as well as OAuth on Android. This section showed you how to create an Android client consuming API's which require Hawk authentication. You've learned how to intercept the request and add the required authorization token based on your credentials.

Honestly, if you read through this complete chapter and understood the key elements for each authentication method, you're absolutely ready for real world scenarios. Don't fear of authenticating your client against APIs!

## Chapter Summary

Within this chapter, you've learned how to authenticate requests appropriately for various authentication methods. Authentication is a very common use case for clients consuming data from APIs and we hope you benefit from all the code examples. We know this chapter is code-heavy and we intentionally provided them, because it's easier to understand that actually there is just a single authentication value between authorized and unauthorized data access. We geeks need some code to understand the context!

You should've learned:

- [x] Use basic authentication with username/email and password
- [x] Authenticate requests with a token value
- [x] Follow the OAuth 2 flow to receive and apply the access token
- [x] How to refresh expired access token

- [x] Apply Hawk credentials to generate the authorization header

Within the next chapter, we show you how to up- and download files with Retrofit and what configurations to keep in mind when working with files.

# Chapter 5 — File Upload & Download

Nowadays, file handling on client-side is an essential capability to enable features like file upload, particularly images. If you're providing a user profile within your Android app, you typically have the ability to change the profile picture. Additionally, your app may have some kind of social stream including images uploaded by users. Within this chapter, we guide you through the challenges of file uploads with Retrofit.

In the second part of this chapter, we'll demonstrate Retrofit's potential to download files from the Internet to the device's storage. This can be used if your app requires some dynamic file data, for example as a .zip file.

## File Upload

Most Android apps need some kind of upload functionality where you can send your profile picture to the server or upload other files to share with your friends. However, this section can be used for any kind of file and not just images.

Retrofit provides the ability to perform multipart requests, which enable clients to upload files to a server. To perform these kind of requests, the `@Multipart` annotation is required on the request declaration.

Before we dive deeper into the file upload topic with Retrofit, let's shortly recap the previously used functionality in v1. Retrofit 1 used a class called `TypedFile` for file uploads to a server. This class has been removed from Retrofit 2. Further, Retrofit now leverages the OkHttp[31] library for any network operation and as a result OkHttp's classes for use cases like file uploads.

Using Retrofit, you need to use either OkHttp's `RequestBody` or `MultipartBody.Part` classes and encapsulate your file into a request body. Let's have a look at the interface definition for file uploads.

```
public interface FileUploadService {
    @Multipart
    @POST("/upload")
    Call<ResponseBody> upload(@Part("description") RequestBody description,
                              @Part MultipartBody.Part file);
}
```

Let us explain each part of the definition above and we're starting with the annotation for description. The description is just a string value wrapped within a RequestBody instance. Secondly,

[31]https://github.com/square/okhttp

there's another @Part within the request: the actual file. We use the `MultipartBody.Part` class that allows us to send the actual file name besides the binary file data with the request. You'll see how to create the file object correctly within the following section.

## Android Client Code

We've defined the necessary interface for Retrofit and now we can move on and touch the Android's part to upload a file. We'll use the `ServiceGenerator` class which generates a service client. We've introduced the `ServiceGenerator` class in the first chapter of this book, which you should be familiar with.

```
Uri fileUri = ... // from a file chooser or a camera intent

FileUploadService service =
        ServiceGenerator.createService(FileUploadService.class);

// use the FileUtils to get the actual file by uri
File file = FileUtils.getFile(this, fileUri);

// create RequestBody instance from file
RequestBody requestFile =
        RequestBody.create(MediaType.parse("multipart/form-data"), file);

// MultipartBody.Part is used to send also the actual file name
MultipartBody.Part body =
        MultipartBody.Part.createFormData(
            "picture",
            file.getName(),
            requestFile);

// add another part within the multipart request
String descriptionString = "hello, this is description speaking";
RequestBody description =
        RequestBody.create(
                MediaType.parse("multipart/form-data"), descriptionString);

// finally, execute the request
Call<ResponseBody> call = service.upload(description, body);
call.enqueue(new Callback<String>() {
    @Override
    public void onResponse(Call<String> call, Response<String> response) {
        Log.v("Upload", "success");
    }

    @Override
    public void onFailure(Call<String> call, Throwable t) {
        Log.e("Upload", t.getMessage());
    }
});
```

The snippet above shows you the code to initialize the payload (`body` and `description`) and how to use the file upload service. As already mentioned, the `RequestBody` class is from OkHttp and used for the description. Its `.create()` method requires two parameters: first, the media type which gets parsed from `multipart/form-data` and second, the actual data.

Besides the description, you'll add the file wrapped into a `MultipartBody.Part` instance. That's what you need to use to appropriately upload files from client-side. Further, you can add the original file name within the `createFormData()` method and reuse it on your backend.

## Remember the Content-Type

Please keep an eye on Retrofit's content type. If you intercept the underlying OkHttp client and change the content type to `application/json`, your server might have issues with the deserialization process. Make sure you're not defining the header indicating you're sending JSON data. The header has to stay as `multipart/form-data`.

## Exemplary Hapi Server for File Uploads

If you already have your backend project, you can lean on the example code below. We use a simple hapi[32] server with a `POST` route available at `/upload`. Additionally, we tell hapi to don't parse the incoming request, because we use a Node.js library called multiparty[33] for the payload parsing.

Within the callback of multiparty's parsing function, we're logging each field to show its output.

[32]http://hapijs.com
[33]https://github.com/andrewrk/node-multiparty

```
method: 'POST',
path: '/upload',
config: {
    payload: {
        maxBytes: 209715200,
        output: 'stream',
        parse: false
    },
    handler: function(request, reply) {
        var multiparty = require('multiparty');
        var form = new multiparty.Form();
        form.parse(request.payload, function(err, fields, files) {
            console.log(err);
            console.log(fields);
            console.log(files);

            return reply(util.inspect({fields: fields, files: files}));
        });
    }
}
```

The Android client expects a return type of `String`, we're sending the received information as response. Of course your response will and should look different :)

Below you can see the output of a successful request and payload parsing. The first `null` is the `err` object. Afterwards, you can see the `fields` which is only the description as part of the request. And last but not least, the file is available within the `picture` field. Here you see our previously defined names on client side. `20160312_095248.jpg` is passed as the original name and the actual field name is `picture`. For further processing, access the uploaded image at `path`'s location.

**Server Log for Parsed Payload**

```
null
{ description: [ 'hello, this is description speaking' ] }
{ picture:
   [ { fieldName: 'picture',
       originalFilename: '20160312_095248.jpg',
       path: '/var/folders/rq/q_m4_21j3lqf1lw48fqttx_80000gn/T/X_sxX6LDUMBcuUcUG\
DMBKc2T.jpg',
       headers: [Object],
       size: 39369 } ] }
```

Now you've learned how to upload a file to a server and pass additional information with it. In the next section, we'll expand on the topic and upload multiple files.

## Upload of Multiple Files

You've seen how to upload a single file. The implementation for more than one file is not that much different. First of all, we need to extend our `FileUploadService` interface for the new upload with multiple files:

```
public interface FileUploadService {
    // previous code for single file uploads
    @Multipart
    @POST("upload")
    Call<ResponseBody> uploadFile(
            @Part("description") RequestBody description,
            @Part MultipartBody.Part file);

    // new code for multiple files
    @Multipart
    @POST("upload")
    Call<ResponseBody> uploadMultipleFiles(
            @Part("description") RequestBody description,
            @Part MultipartBody.Part file1,
            @Part MultipartBody.Part file2);
}
```

If you compare the two interface methods for `uploadFile()` and `uploadMultipleFiles()`, you can spot the difference very easily. Retrofit makes it simple to add new file parts. On the side of the interface declaration, we just need to add another line `@Part MultipartBody.Part file`. The `uploadMultipleFiles()` method currently only supports exactly two files. You need to adjust the amount of `MultipartBody.Part` parts to your needs.

**Dynamic Amount of Files**

Currently, Retrofit 2.0 does not support a dynamic amount of files. We'll update the book once this is possible.

Declaring the interface is only half of the work. The other part is the implementation in the activity or fragment lifecycle. Since we're working with multiple files now, we've implemented two helper methods to make things more robust:

```java
public static final String MULTIPART_FORM_DATA = "multipart/form-data";

@NonNull
private RequestBody createPartFromString(String descriptionString) {
    return RequestBody.create(
            MediaType.parse(MULTIPART_FORM_DATA), descriptionString);
}

@NonNull
private MultipartBody.Part prepareFilePart(String partName, Uri fileUri) {
    // use the FileUtils to get the actual file by uri
    File file = FileUtils.getFile(this, fileUri);

    // create RequestBody instance from file
    RequestBody requestFile =
        RequestBody.create(MediaType.parse(MULTIPART_FORM_DATA), file);

    // MultipartBody.Part is used to send also the actual file name
    return MultipartBody.Part.createFormData(partName, file.getName(), requestFi\
le);
}
```

The `createPartFromString()` method can be useful for sending descriptions along a multipart upload. The `prepareFilePart()` method builds a `MultipartBody.Part` object, which contains a file (like a photo or video). The wrapping in a `MultipartBody.Part` is necessary to upload files with Retrofit.

So let's use our helper methods to send two files:

```java
Uri file1Uri = ... // get it from a file chooser or a camera intent
Uri file2Uri = ... // get it from a file chooser or a camera intent

// create upload service client
FileUploadService service =
        ServiceGenerator.createService(FileUploadService.class);

// create part for file (photo, video, ...)
MultipartBody.Part body1 = prepareFilePart("video", file1Uri);
MultipartBody.Part body2 = prepareFilePart("thumbnail", file2Uri);

// add another part within the multipart request
RequestBody description = createPartFromString("hello, this is description speak\
ing");
```

```

// finally, execute the request
Call<ResponseBody> call = service.uploadMultipleFiles(description, body1, body2);
call.enqueue(new Callback<ResponseBody>() {
    @Override
    public void onResponse(Call<ResponseBody> call,
            Response<ResponseBody> response) {
        Log.v("Upload", "success");
    }

    @Override
    public void onFailure(Call<ResponseBody> call, Throwable t) {
        Log.e("Upload error:", t.getMessage());
    }
});
```

That's all you need to do to send multiple files in one request. Of course, you could add another or many more parts to the interface, if necessary. Using this approach you can send as many files as you wish.

In the next section, we'll demonstrate a way on how to pass a dynamic amount of information along the file using `@PartMap`.

## Passing Multiple Parts Along a File with @PartMap

In the previous section, we've shown you how to upload files and upload multiple files. So far, we've focused on the file part of multipart requests. In this section, we'll concentrate on the data that goes along with the request, for example description string(s).

Multipart requests are often used for forms with an additional file. For example, we've utilized it in the past for a feedback form, which also allows the user to upload a photo.

If you just need to pass a single or two descriptions with a file, you can just declare it as a `@Part` in your service declaration:

```
public interface FileUploadService {
    // previous code for single description
    @Multipart
    @POST("upload")
    Call<ResponseBody> uploadFile(
            @Part("description") RequestBody description,
            @Part MultipartBody.Part file);
}
```

This works great for small use cases, but if you need to send more than a handful of properties, it gets quite messy, especially if not all of them are always set.

Retrofit offers an easy solution, which makes the uploads quite customizable: `@PartMap`. `@PartMap` is an additional annotation for a request parameter, which allows us to specify how many and which parts we send during runtime. This can very helpful if your form is very long, but only a few of those input field values are actually send. Instead of declaring an interface method with 20 or more parameters you can use a single `@PartMap`. Let's see this in action!

First of all, we need to create a new interface method with the `@PartMap` annotation:

```
public interface FileUploadService {
    // declare a description explicitly
    // would need to declare
    @Multipart
    @POST("upload")
    Call<ResponseBody> uploadFile(
            @Part("description") RequestBody description,
            @Part MultipartBody.Part file);

    @Multipart
    @POST("upload")
    Call<ResponseBody> uploadFileWithPartMap(
            @PartMap() Map<String, RequestBody> partMap,
            @Part MultipartBody.Part file);
}
```

It's important that you use a `Map<String, RequestBody>` implementation as a parameter type for the `@PartMap` part of the request. As we've explained above, this allows you to send a runtime-dependent list of data along with your file. The brackets after the `PartMap` are optional. You need to use them, if you want to specify the encoding, similar to the content encoding in FieldMap[34]s.

The second part is filling the Map with data. In the previous section, we've introduced a helper method to create a `RequestBody` for a String variable:

[34]https://futurestud.io/blog/retrofit-send-data-form-urlencoded-using-fieldmap/

```
public static final String MULTIPART_FORM_DATA = "multipart/form-data";

@NonNull
private RequestBody createPartFromString(String descriptionString) {
    return RequestBody.create(
            MediaType.parse(MULTIPART_FORM_DATA), descriptionString);
}
```

Finally, let's use this method and view the entire code from creating the Retrofit service, to filling the request with data and enqueuing the request:

```
Uri fileUri = ... // from a file chooser or a camera intent

// create upload service client
FileUploadService service =
        ServiceGenerator.createService(FileUploadService.class);

// create part for file (photo, video, ...)
MultipartBody.Part body = prepareFilePart("photo", fileUri);

// create a map of data to pass along
RequestBody description = createPartFromString("hello, this is description speak\
ing");
RequestBody place = createPartFromString("Magdeburg");
RequestBody time = createPartFromString("2016");

HashMap<String, RequestBody> map = new HashMap<>();
map.put("description", description);
map.put("place", place);
map.put("time", time);

// finally, execute the request
Call<ResponseBody> call = service.uploadFileWithPartMap(map, body);
call.enqueue(...);
```

Of course, depending on your use case you've to fill in a bit of logic. If you've a similar scenario to our feedback form, you could go through the list of `EditTexts` and only add the content of non-empty ones to your multipart request.

We've shown you the trick of the `PartMap` annotation. Retrofit makes it easy to counter overblowingly long method declarations of multipart requests by offering to send multiple parts with `@PartMap`.

File uploads and passing data with it are an essential feature within up-to-date apps and it's important that you implement it in a clean way. The download of files can be essential to some apps as well. In the next section, we'll look at a file download implementation based on Retrofit.

## File Download

File download with Retrofit is one of the most requested topics on Future Studio. In this section, we'll give you all the insights and snippets you need to use Retrofit to download everything, from tiny `.png`'s to large `.zip` files.

### How to Specify the Retrofit Request

If you're reading the book just for this section and you haven't written code for any Retrofit requests yet, please check our previous chapters to get started. For all you Retrofit experts: the request declaration for downloading files looks almost like any other request:

```
public interface DownloadService {
    // option 1: a resource relative to your base URL
    @GET("/resource/example.zip")
    Call<ResponseBody> downloadFileWithFixedUrl();

    // option 2: using a dynamic URL
    @GET
    Call<ResponseBody> downloadFileWithDynamicUrlSync(@Url String fileUrl);
}
```

If the file you want to download is a static resource (always at the same spot on the server) and on the server your base URL refers to, you can use option 1. As you can see, it looks like a regular Retrofit request declaration. Please note, that we're specifying `ResponseBody` as return type. You should not use anything else here, otherwise Retrofit will try to parse and convert it, which doesn't make sense when you're downloading a file.

The second option is new to Retrofit. You can now easily pass a dynamic value as full URL to the request call. This can be especially helpful when downloading files, which are dependent of a parameter, user or time. You can build the URL during runtime and request the exact file without any hacks.

Pick what kind of option is useful to you and move on to the next section.

### How to Call the Request

After declaring our request, we need to actually call it:

```
FileDownloadService downloadService =
    ServiceGenerator.create(FileDownloadService.class);

Call<ResponseBody> call =
    downloadService.downloadFileWithDynamicUrlSync(fileUrl);

call.enqueue(new Callback<ResponseBody>() {
    @Override
    public void onResponse(
            Call<ResponseBody> call,
            Response<ResponseBody> response) {
        if (response.isSuccess()) {
            Log.d(TAG, "server contacted and has file");

            boolean writtenToDisk = writeResponseBodyToDisk(
                response.body(),
                null);

            Log.d(TAG, "file download was a success? " + writtenToDisk);
        } else {
            Log.d(TAG, "server contact failed");
        }
    }

    @Override
    public void onFailure(Call<ResponseBody> call, Throwable t) {
        Log.e(TAG, "error");
    }
});
```

Once we've created the service, we'll make the request just like any other Retrofit call! There is just one thing left, which currently hides behind the function `writeResponseBodyToDisk()`: writing the file to the disk!

## How to Save the File

The `writeResponseBodyToDisk()` method takes the `ResponseBody` object and reads and writes the byte values of it to the Android's disk. The code looks much more difficult than it actually is:

```
private boolean writeResponseBodyToDisk(ResponseBody body) {
    try {
        // todo change the file location/name according to your needs
        File futureStudioIconFile =
            new File(getExternalFilesDir(null) +
            File.separator +
            "Future Studio Icon.png");

        InputStream inputStream = null;
        OutputStream outputStream = null;

        try {
            byte[] fileReader = new byte[4096];

            long fileSize = body.contentLength();
            long fileSizeDownloaded = 0;

            inputStream = body.byteStream();
            outputStream = new FileOutputStream(futureStudioIconFile);

            while (true) {
                int read = inputStream.read(fileReader);

                if (read == -1) {
                    break;
                }

                outputStream.write(fileReader, 0, read);

                fileSizeDownloaded += read;

                Log.d(TAG, "download " + fileSizeDownloaded + "/" + fileSize);
            }

            outputStream.flush();

            return true;
        } catch (IOException e) {
            return false;
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }

            if (outputStream != null) {
                outputStream.close();
            }
        }
    } catch (IOException e) {
        return false;
    }
}
```

Most of it is just regular Java I/O boilerplate. You might need to adjust the first line where the location and naming of your file is being saved. When you have done that, you're ready to download files with Retrofit!

But we're not completely ready for all files yet. There is one major issue: by default, Retrofit puts the entire server response into memory before processing the result. This works fine for some JSON or XML responses, but large files can easily cause Out-of-Memory-Errors.

If your app needs to download even slightly larger files, we strongly recommend reading the next section.

## Beware with Large Files: Use @Streaming!

If you're downloading a large file, Retrofit would try to move the entire file into memory. In order to avoid that, we've to add a special annotation to the request declaration:

```
@Streaming
@GET
Call<ResponseBody> downloadFileWithDynamicUrlAsync(@Url String fileUrl);
```

The `@Streaming` declaration doesn't mean you're watching a Netflix file. It means that instead of moving the entire file into memory, it'll pass along the bytes right away. But be careful, if you're adding the `@Streaming` declaration and continue to use the code above, Android will trigger a `android.os.NetworkOnMainThreadException`.

Thus, the final step is to wrap the call into a separate thread, for example in a lovely ASyncTask:

```
final FileDownloadService downloadService =
                ServiceGenerator.create(FileDownloadService.class);

new AsyncTask<Void, Long, Void>() {
   @Override
   protected Void doInBackground(Void... voids) {
           Call<ResponseBody> call = downloadService.downloadFileWithDynamicUrlSync(fil\
eUrl);
           call.enqueue(new Callback<ResponseBody>() {
               @Override
               public void onResponse(
                       Call<ResponseBody> call,
                       Response<ResponseBody> response) {
                   if (response.isSuccess()) {
                       Log.d(TAG, "server contacted and has file");

                       boolean writtenToDisk =
                           writeResponseBodyToDisk(response.body(), null);

                       Log.d(TAG, "file download was a success? " + writtenToDisk);
                   }
                   else {
                       Log.d(TAG, "server contact failed");
                   }
               }
       return null;
   }
}.execute();
```

If you remember the `@Streaming` declaration and this snippet, you can download even large files with Retrofit efficiently.

**Retrofit is not build for file downloads**

Retrofit's purpose and design are not suited for file downloads. While it is possible, in many cases it's not the clean way to implement file downloads. An interesting alternative is the Android native DownloadManager[35].

[35]http://developer.android.com/reference/android/app/DownloadManager.html

## Chapter Summary

Retrofit offers out of the box support for multipart request parameters, including files. We showed you how to define your service interface and what annotations you need to use for file upload. Don't forget to keep the content-type in mind. It's that minor detail that can cost you hours (we're talking from experience) and does the trick when uploading files correctly from your Android app. Additionally, you've seen how to download files with Retrofit. In order to ensure you got the full value out of this chapter, review if you've learned all of its content:

- [x] How to upload files
- [x] Be careful with the Content-Type and interceptors
- [x] How to upload multiple files
- [x] When and how to use PartMap for a dynamic amount of multipart parameters
- [x] How to download files
- [x] When to use the `@Streaming` annotation

This chapter guided you through the essentials on file handling using Retrofit. Within the next chapter, you'll learn how to use pagination with Retrofit for different API implementations. We'll walk you through the fundamentals on how to request small chunks from the overall resource without downloading massive amount of data.

# Chapter 6 — Pagination on Android

Pagination is a complex topic that doesn't follow a given specification. The server-side implementation varies throughout different APIs and you need to adapt your client-side code by following the related API documentation.

This chapter will go in detail on three pagination implementations using Retrofit. If you need pagination within your Android app, there might be a fit for you:

**Pagination Types**

1. Query Parameter
2. Response header and dynamic URLs (like GitHub)
3. Range headers (like Heroku)

Even if you need to implement any other pagination style for your app, there'll be benefits within this guides that help you to get going more quickly.

## Pagination Response Status

Depending on the API implementation, you'll receive a **response status** of either `200` or `206`. Particularly the `206 Partial Content` response is more suitable as you're receiving just parts of the overall content in the form of a data subset. However, as you're just consuming the information on client-side, a response status within the `2xx` range is fine. If you heavily rely on status codes, check the related API documentation on how to handle response codes properly.

## Pagination Using Query Parameter

Requesting data from API endpoints that might respond with large datasets is obviously bad practice. Breaking down massive information packets into smaller chunks is the idea behind pagination. There's no de facto standard on how to implement pagination on server-side and therefore, clients have to adapt to the solution chosen by API developers.

This is the first guide on how to implement a pagination strategy on Android to consume a partial dataset with each request for a given endpoint. You'll learn the fundamental ideas behind pagination using query parameters and what pitfalls to watch during your implementation. In future articles, you'll learn how to tackle pagination on Android using HTTP response header information with dynamic URLs and the approach of leveraging range headers.

## Pagination via Query Parameter

You might have a basic understanding about the idea behind pagination using query parameter. We're seeing the application of this pagination type throughout the web: on blogs, news sites, etc. Mainly, this is because we prevent information overload when visiting any web page. Let's take a blog as an example. When publishing the first post, there's still a good overview of all the content (it's just one post). The page loads fast and users can easily snatch through the given information. Imagine the situation after one year of consistently publishing one article each week. Your blog's startpage will have a lot of posts and keep in mind that all the information needs to be transferred to the client's device.

That's the reason why all blogs break down the content into various pages and you can scan the posts straight forward and read the next page (if you're still interested).

The same scenario applies to APIs that only send a selected chunk of data instead of sending the complete resource set available. Requesting an API endpoint for data will result in receiving just partial content as payload. You need to request the second, or third, …, or n-th page to have all data available on client-side.

### What Responses to Expect

You already know that the actual server-side implementation forces you to follow their rules on how to receive the data.

Let's assume you're executing the following request:

```
GET
https://your.api.url/pagination?page=1
```

Depending on the response for your request, you might get metadata including pagination information, like the actual resource size, available pages, next page, previous page, last page, etc.

**Possible Response Payload #1**

```
{
  pagination: {
    currentPage: 1,
    nextPage: 2,
    lastPage: 17,
    itemCount: 165,
    itemsPerPage: 10
  },
  items: [
    … // list if items for current page
  ]
}
```

In contrast, there are existing APIs that don't provide any meta information about the available resources. This way, you can't compare the current page of the response with your saved page on client-side and decide whether you can stop requesting data or not. You need to rely on the response status to avoid an infinite request loop. Precisely, if you're requesting a page outside the resource scope, the API without pagination metadata should send an appropriate response status, like `404` so that you know there's no more data to fetch.

**Possible Response Payload #2**

```
{
  [
    … // list if items for current page
  ]
}
```

As you can see within the exemplary response payload above, there's either pagination information within the HTTP response headers or nothing available at all. If you don't have any pagination data, your server companion hopefully let's you know once you're outside the accepted pages range.

## Request the First Dataset

You've learned about some of the pagination pitfalls within the previous section. That doesn't hold us back from implementing query parameter based pagination on Android using Retrofit.

**Starting Point** In the following, you need to assume there's an imaginary API where you're going to fetch items at the API endpoint URL: `https://your.api.url/pagination`. You're going to receive partial responses without any pagination information within the response headers or payload. Once you request a page where no data is available anymore, you'll receive a `404` response status.

The developer of this imaginary API obviously has the opinion, that it's fine to have a one request trade-off to determine when you've all existing data available on client-side.

Finally, get ready to put your hands to the keyboard. Based on the description above, you'll implement the functionality to request partial subsets of data from the API endpoint that ultimately accumulate to the complete result set. The `PaginationService` from the snippet below describes the service endpoint. We're using the `page` query parameter of type `Integer` to request the data for a given page.

```
public interface PaginationService {
    @GET("pagination")
    Call<List<PaginationItem>> fetchPage(@Query("page") Integer page);
}
```

As already mentioned, you'll receive a list of items for a given page as the response payload for your request. The focus of this chapter is on the pagination part, that's the reason we keep the class for individual items quite simple. Each item object is represented by the `PaginationItem` class.

```java
public class PaginationItem {

    private String name;

    public PaginationItem() {
    }

    public String getName() {
        return name;
    }
}
```

At this point, you know the fundamentals of pagination using query parameters and you've created the required service interface including the data transfer object, `PaginationItem`. The code below outlines the skeleton for the `QueryParamPaginationActivity`. The activity just has the `service` client as a class property. The service client is initialized once within the `onCreate()` method so that's available within other methods of the activity. Further, we're starting the request to fetch the initial dataset as well by calling the `fetchPage()` method.

```java
public class QueryParamPaginationActivity extends AppCompatActivity {

    private PaginationService service;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pagination);

        // initialize service client
        service = ServiceGenerator.createService(PaginationService.class);

        // fetch first dataset
        fetchPage(1);
    }

    private void fetchPage(final int page) {
        Call<List<PaginationItem>> call = service.fetchPage(page);
        call.enqueue(new Callback<List<PaginationItem>>() {
            @Override
            public void onResponse(Call<List<PaginationItem>> call,
                                   Response<List<PaginationItem>> response) {
                if (response.isSuccessful()) {
```

```
                // use requested items
                List<PaginationItem> body = response.body();

                // fetch data from next page
                fetchPage(page + 1);
            }
        }

        @Override
        public void onFailure(Call<List<PaginationItem>> call, Throwable t) {

        }
    });
  }
}
```

Actually, the implementation is sort of simplified and requesting the next dataset is initiated within the response callback. Usually, you're not going to request all available data from the API endpoint by starting the request for the next page directly after receiving the current data subset. Common scenarios to request the next page of data are like scrolling to the end of a list view or by clicking a button to "load more".

Within the snippet above, you see that the `fetchPage(int page)` has the page value as method parameter. You might need to adjust that and store the page value within a class property, because you don't receive any pagination metadata from the API and if you're not requesting the next page immediately after a successful response, you need to keep the current client state.

## Fetch Next Dataset

Fetching data in paginated fashion is kind of straight forward and as soon as you notice a given event, like a list view reaches the end or a user clicks a "load more" button, you're starting to fetch the next dataset. Use the `fetchPage(int page)` method to request all individual datasets. Actually, there's nothing special to consider, because the data is chunked in blocks of `x` items (depending on the API) and you need to request each page consecutively to get the desired information.

## When to Stop Requesting the Next Page

This is completely dependent on the API and your client-side implementation on Android. If you're receiving pagination information including the number of pages, you can compare your current state and check whether you've already landed the eagle. If you don't get metadata, you need to check if you're receiving a response status code like `4xx` to stop requesting further data. You get the point and understand that there are multiple variables to keep in mind when paginate through the data of an API endpoint.

When using pagination with query parameters, you have to implement most of the client-side logic yourself. You need to keep track of the current state and what page was requested lastly. The upcoming section shows you a promiment API that helps you out and provides the required urls to paginate along the resource using HTTP response headers.

## Pagination Using Link Header and Dynamic Urls (Like GitHub)

Within this second section, you'll learn how to implement pagination with Retrofit using HTTP response header information generated by the server and dynamic urls to fetch desired parts to complete the dataset on client-side.

### Prerequisites

There's a quite well-known API that uses a response header for pagination: GitHub. Precisely, we'll request data using the GitHub API and make use of the given header field that contain any required information. If you don't have any clue on how to traverse the GitHub API with pagination[36], you can read the linked guide to get an idea.

### Pagination Using the `Link` Header

Let's look at a concrete example: request the list of repositories for a given GitHub user. You can use the GitHub API endpoint that is located at `https://api.github.com/users/{username}/repos` to fetch (a paginated) list of repositories for the individual `{username}`.

The `Link` response header consists of four different parts. Depending on the current request and response state, there are not all four parts available. We'll look at an example in a second, that will clarify all the things :)

**Four parts of `Link` header**

1. `next`: link to the next data subset
2. `prev`: link to the previous data subset
3. `first`: link to the first data subset
4. `last`: link to the last data subset

Alright, to make things approachable, we'll use `square` as the username and execute a request to fetch the list of Square's repositories from GitHub's API. We're starting out using the following request type and url:

[36]https://developer.github.com/guides/traversing-with-pagination/

```
GET
https://api.github.com/users/square/repos
```

Actually, you can use tools like Postman[37] to test API requests with ease. Postman lets you freely compose requests and access response headers and payload!

Depending on the current request state and if you've executed any prior request, the `Link` response header looks different. When fetching the first data set, you'll receive a response only containing the header parts for `next` and `last`, because it's actually the `first` data test and there's no `prev` one. The response header looks like this:

```
Link: <https://api.github.com/user/82592/repos?page=2>; rel="next",
  <https://api.github.com/user/82592/repos?page=6>; rel="last"
```

Did you recognize the url change? Due to the fact that GitHub generates the links for further data subsets on server-side, trust in the correctness of the received links and use them for further requests, even though the path parameter varies.

Later in this section, you'll learn how to parse the received `Link` header information. For now, keep focus on the different values within the header for the individual requests.

When requesting the second page, you'll receive all four parts of the `Link` response header including a value for each part. The link header will look like this:

```
Link: <https://api.github.com/user/82592/repos?page=3>; rel="next",
  <https://api.github.com/user/82592/repos?page=6>; rel="last",
  <https://api.github.com/user/82592/repos?page=1>; rel="first",
  <https://api.github.com/user/82592/repos?page=1>; rel="prev"
```

**Github Uses Status Code `200` for Partial Content**

Just as a side note, GitHub's API returns response status `200` for partial content.

## Request the First Data Set

If you read the previous section describing the pagination approach using the `Link` header, you've already seen the request against the GitHub API to fetch the list of repositories for a given user. In the following, we'll use the GitHub API for further requests to illustrate the pagination implementation using Retrofit.

Therefore, we need to declare a service interface that consists of two methods. The following `GitHubService` outlines the required methods for the initial request and to fetch further results from various pages.

[37]https://www.getpostman.com/

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

    @GET
    Call<List<GitHubRepository>> reposForUserPaginate(@Url String url);
}
```

The `reposForUser` method is used for the initial request without any pagination information. We have to provide a username that will be used as a path parameter within the request url. You know, Retrofit maps the given path parameter automatically.

The second method `reposForUserPaginate` just takes a url as a parameter. Retrofit maps the url parameter dynamically as the request url and that allows us to fetch further data without defining any additional query or path parameter. Convenient, right?! :)

Up to this point, you've learned the foundations about pagination using the `Link` response header and we've already declared the required service interface to fetch multiple partial repository sets for a given GitHub user. To map the received data to Java objects, we'll use the following `GitHubRepository` class. GitHub is sending a lot more information than just the repository name. The class below is just illustrative and we're actually more interested in paginated requests than proper data mapping.

```java
public class GitHubRepository {
    private String name;

    public GitHubRepository() {
    }

    public String getName() {
        return name;
    }
}
```

We have the basics in place, it's time to put things to work and compose an activity where we'll fetch the complete list of repositories for a given user in paginated fashion. The code below outlines the skeleton for the `GitHubPaginationActivity` and looks much more complicated than it actually is. As you can see, there are two class fields: the `service` client and `ghCallback` which is used as the callback implementation for the individual requests to fetch the next set of repositories.

Actually, we're initiating the next request to fetch the next page of repositories directly within the callback. Adjust the implementation to your needs, like fetching the next data set when scrolling to the end of a list view or when pressing a button to load more data.

```java
public class GitHubPaginationActivity extends AppCompatActivity {

    private GitHubService service;
    private Callback<List<GitHubRepository>> ghCallback =
            new Callback<List<GitHubRepository>>() {
        @Override
        public void onResponse(Call<List<GitHubRepository>> call,
                               Response<List<GitHubRepository>> response) {
            if (response.isSuccess()) {
                // use list of requested GitHub repos
                List<GitHubRepository> body = response.body();

                fetchReposNextPage(response);
            } else {
                Log.e("Request failed: ", "Cannot request repositories");
            }
        }

        @Override
        public void onFailure(Call<List<GitHubRepository>> call,
                              Throwable t) {
            Log.e("Error fetching repos", t.getMessage());
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_github_pagination);

        // Change base url to GitHub API
        ServiceGenerator.changeApiBaseUrl("https://api.github.com/");

        // Create a simple REST adapter which points to GitHub’s API
        service = ServiceGenerator.createService(GitHubService.class);

        // Fetch and print a list of repositories for user “square”
        // “fs-opensource” has too little repositories for pagination ;-)
        Call<List<GitHubRepository>> call = service.reposForUser("square");
        call.enqueue(ghCallback);
    }

    // other methods
}
```

Within the `onCreate()` method, we use the previously introduced `ServiceGenerator` class to create a service client based on the `GitHubService` interface. If you're not familiar with the service generator, please read the section on how to create a sustainable Android client within the getting started guide of this series. Furthermore, we've introduced the functionality to dynamically adjust Retrofit's base url within the request chapter. We're leveraging this functionality to change the base url to GitHub's API for the requests within this activity.

Once we instantiated the service client, there's nothing holding us back to request the first partial set of repository data. To be honest, we'd love to show the functionality using our `fs-opensource`[38] user, but we don't have sufficient repositories available at the time of writing this book to show the functionality of pagination. That's the reason why we fetch the repositories of `square`.

## Parse Pagination Information From Response Header

At this point, we've executed the initial request to fetch the first subset of data and received the response that includes the pagination details within the `Link` header. We recommend to have a utility class within your project that parses the response header information for pagination. You don't need to implement that on your own. Benefit from the existing PageLinks[39] class that does the job very well.

We've applied minor adjustments to parse the response headers from Retrofit instead of a specific GitHub response class. With the help of the utility class, you're able to fetch the next set of data easily.

## Fetch Data From Next Page

This method extends the introduced `GitHubPaginationActivity` with the functionality to request the next data set (if there's more to fetch). Within this method, make use of the adjusted `PageLinks` class that we call `GitHubPagelinksUtils`. The only functionality of that utils class is to parse the `Link` header information. If there's an existing link for the `next` meta keyword, you can move on and request the next page of data. If you've already requested the last page or there isn't any further data to request, the `next` link will be `null` and there's nothing do to.

[38]https://github.com/fs-opensource

[39]https://github.com/eclipse/egit-github/blob/master/org.eclipse.egit.github.core/src/org/eclipse/egit/github/core/client/PageLinks.java

```
private void fetchReposNextPage(Response<List<GitHubRepository>> response) {
    GitHubPagelinksUtils pagelinksUtils =
            new GitHubPagelinksUtils(response.headers());
    String next = pagelinksUtils.getNext();

    if (TextUtils.isEmpty(next)) {
        return; // nothing to do
    }

    Call<List<GitHubRepository>> call = service.reposForUserPaginate(next);
    call.enqueue(ghCallback);
}
```

This method is also the place where our second service method is used. Due to the fact that GitHub already sends the entire url for the next data set, we can directly use it and pass the value as a dynamic url parameter to the `service.reposForUserPaginate()` method. Awesome!

You can see that the `ghCallback` is used again, because we're expecting the same response: a list of repositories. The defined callback already has the functionality implemented that should be applied to the response.

That's all the magic behind paginated requests using header information and dynamic urls. Actually, using the introduced functionality in Retrofit to apply dynamic urls for individual endpoints enables a convenient way to paginate through various pages for a resource set.

In the following section, you'll learn about another pagination type that heavily uses HTTP request and response headers to fetch partial content for given requests.

## Pagination Using Range Header Fields (Like Heroku)

We've previously guided you through the details of partial responses with pagination using query parameter and response header including dynamic urls. This third part around pagination will show you how to implement the functionality on client-side using HTTP range header fields to fetch partial data from a resource set. Usually, the range headers are already prepared by the server and you can adjust them on your client.

### Prerequisites

Within this section, we'll go into detail on range header fields within the HTTP response. An exemplary API that implements pagination using range headers is Heroku. We'll use the Heroku API to execute various requests that illustrate the usage of all range header fields. If you don't have any clue about range headers on Heroku, you can read the section on Heroku Ranges[40] within their platform reference.

[40] https://devcenter.heroku.com/articles/platform-api-reference#ranges

## Pagination via Range Header Fields

We need to go through the basics of the idea behind HTTP range headers for pagination before moving on to the more practical part. You've probably recognized, that we use the plural "headers" all the time. That's correct, we have to keep an eye on multiple header values, all with different functionalities.

The important thing to understand when using pagination with range headers is that you specify a range of values or a specific subset by using elements from set theory. Please don't think of the idea behind the ranges in a page-like style. You need to think of it as elements within a subset that ranges from `a` to `b` for column `c` and that's enclosed by given tokens or characters. We'll get practical later within this section :)

When executing requests, we can manipulate the partial data returned by adding a request header called `Range` and assign the desired value. Actually, that's the only thing required to request data and use filters like a specific order, limit the number of items within the response, etc.

**Request Header**

- `Range`: used to customize the response data (range by identifier, order, number of items, etc)

In contrast to the `Range` request header, we'll receive one or many response headers with further information about the available resource, how to request the next subset and what fields can be used to order the results by ranges.

**Response Headers**

- `Accept-Ranges`: list of columns that can be used to order the response
- `Content-Range`: has a format like `identifier start..end` and indicates the current response range
- `Next-Range`: value that can be used to request the next dataset

The following snippet is an exemplary excerpt from Heroku's response range headers:

```
HTTP/1.1 206 Partial Content
Accept-Ranges: id, name, updated_at
Content-Range: name start_project..start_project
Next-Range: name ]stop_projectname..; max=1
```

Also, the accepted ranges are located within the `Accept-Ranges` field to indicate that you can fetch partial datasets from the overall resources for the identifiers `id`, `name` and `updated_at`. The content range depicts the actual value range for the current response. The next range contains the value that you can use for the upcoming request which needs to be set to the `Range` header. The given value of `name ]start_projectname..; max=1` illustrates the usage of tokens from set theory to exclude the previously fetched item(s) and also limits the response size for the next request to one.

Depending on the API implementation, you'll receive a **response status** of either `200` or `206`. Particularly the `206` response is more suitable as you're receiving partial content in the form of a data subset. However, as you're just consuming the information on client-side, a response status within the `2xx` range is fine.

## Request the First Dataset

Within the previous sections you've learned the basics on pagination using range headers. To make things approachable, we'll use Heroku's API to illustrate a client implementation for range headers on Android using Retrofit.

**Heroku Authentication** The API endpoint we're going to call requires authentication. Get your API key within Heroku's dashboard[41]. To authenticate a request, add the `Authorization` header using the API key: `Authorization: Bearer <your API key>`.

We'll request a list of applications that we've created on Heroku and therefore, we need to declare a service interface that maps the API endpoint including required HTTP headers and parameters. The following `HerokuService` outlines the method that we'll use to request the list of applications.

Please note that you need to add authentication. We're using an OkHttp interceptor to add the `Authorization` header for any request. However, you can also add the `Authorization` header as second method parameter and let Retrofit map the field accordingly.

```java
public interface HerokuService {
    @GET("apps")
    @Headers("Accept: application/vnd.heroku+json; version=3")
    Call<List<HerokuApp>> getApps(@Header("Range") String range);
}
```

The `getApps()` method can and will be reused for additional requests to paginate through the various sets of Heroku applications. You need to adjust the individual value for the given `range` header for each request. We'll guide you through each step to fetch all items available.

At this point, you've gained knowledge about the fundamentals of pagination using range headers and we've also declared the required service interface to request the list of applications for our account on Heroku. To map the received data, we need to define a Java class that can be used by Gson (or any other JSON converter) to map the JSON response. Due to the focus on pagination, we'll keep the Java class as minimal as possible. The code snippet below shows the `HerokuApp` class that we'll use for the response mapping.

[41]https://dashboard.heroku.com/account

```
public class HerokuApp {

    private String name;

    public HerokuApp() {
    }

    public String getName() {
        return name;
    }
}
```

Alright, you’ve everything in place to get practical and create an activity that requests your list of Heroku projects. If you don’t have many projects on Heroku, don’t worry! You can limit the amount of projects received within the response so that even two Heroku projects allow you to execute two requests including pagination.

The code below outlines the skeleton for the `HerokuPaginationActivity` and looks a lot more complicated than it actually is. As you can see, there are two class fields: the `service` client and `callback` which is used as the callback implementation for the individual requests to fetch the next set of projects.

The next request to fetch further project data is initiated within the callback. You might want to adjust this implementation and fetch the next dataset when scrolling to the end of a list view or when pressing a button to load more data.

```
public class HerokuPaginationActivity extends AppCompatActivity {

    private HerokuService service;

    private Callback<List<HerokuApp>> callback = new Callback<List<HerokuApp>>()\
 {
        @Override
        public void onResponse(Call<List<HerokuApp>> call,
                               Response<List<HerokuApp>> response) {
            if (response.isSuccess()) {
                List<HerokuApp> body = response.body();
                // use list of request apps

                fetchAppsNextPage(response);
            } else {
                Log.e("Request failed: ", "Cannot request Heroku apps");
            }
        }

        @Override
        public void onFailure(Call<List<HerokuApp>> call, Throwable t) {
            Log.e("Error fetching apps", t.getMessage());
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_github_pagination);

        // Change base url to Heroku API
        ServiceGenerator.changeApiBaseUrl("https://api.heroku.com/");

        // Create a simple REST adapter which points to GitHub's API
        service = ServiceGenerator.createService(HerokuService.class);

        // Fetch and print a list of apps
        Call<List<HerokuApp>> call = service.getApps("name ..; max=1");
        call.enqueue(callback);
    }

    // other methods
}
```

Within the `onCreate()` method, we use the previously introduced `ServiceGenerator` class to create a service client based on the `HerokuService` interface. If you're not familiar with the service generator, please read the section on how to create a sustainable Android client within the getting started guide of this series. Furthermore, we've introduced the functionality to dynamically adjust Retrofit's base url within the request chapter. We're leveraging this functionality to change the base url to Heroku's API for requests within this activity.

At this point, we're ready to perform the initial request and fetch the first set of projects. To be honest, we only have 4 projects associated to our Heroku account. Executing the request without limiting the number of items within the response wouldn't allow us to paginate at all. That's the reason why we add the limitation `max=1` to the `Range` header value within the request. You can see, that we order the results by `name` and don't specify any range: `name ..`. The limitation needs to be separated from the range by a semicolon. That said, the `Range` header for the initial request results in `name ..; max=1`.

## Parse Content-Range Information From Response

You've executed the initial request to fetch the first subset with project data. The response headers include all necessary information about the pagination options. The following snippet illustrates only the related header fields to pagination with their corresponding values:

```
HTTP/1.1 206 Partial Content
Accept-Ranges: id, name, updated_at
Content-Range: name start_project..start_project
Next-Range: name ]stop_projectname..; max=1
```

To just fetch the next partial dataset, you can directly use the given value from the `Next-Range` header as your value for the `Range` request header. If you want to request another range than by `name`, can grab any option from the `Accept-Ranges` header. If you expect overlapping results, you can use the value within `Content-Range` to compare previous results with the current partial content.

## Fetch the Next Dataset

The method within the following code snippet should be part of the previously introduced `Heroku-PaginationActivity`. It includes the functionality to request the next block of Heroku projects. The interesting part is to get the value from `Next-Range` response headers. It contains the information that is required to fetch the next dataset using the already defined filter. If you've already requested the last range or there isn't any further data to request, the `Next-Range` value will be `null` and there's nothing do to.

```
private void fetchAppsNextPage(Response<List<HerokuApp>> response) {
    Headers headers = response.headers();
    String next = headers.get("Next-Range");

    if (TextUtils.isEmpty(next)) {
        return; // nothing to do
    }

    Call<List<HerokuApp>> call = service.getApps(next);
    call.enqueue(callback);
}
```

We've mentioned earlier, that we'll reuse the `getApps()` method for paginated requests. You know that the `Next-Range` header already contains the correct information that can be applied to request the next subset. We can directly pass its value to the `getApps()` method. Retrofit will do the work and map the `Range` header with the given value.

You can see that the `callback` is used again, because we're expecting the same response: a list of projects. The defined callback already has the functionality implemented that should be applied to the response.

**Simplified Implementation**

Please notice, the implementation above is kind of simplified and you should ajudst the part of fetching the next dataset within the callback to avoid a loop of request for large resource sets. Common use cases are like fetching the next dataset when scrolling to the end of a list view or when pressing a button to load more data.

That's all the magic behind paginated requests using range headers. The general idea behind this type of pagination is based on header values and Retrofit ships with solid support to add custom header fields to your requests.

## Chapter Summary

Pagination is a common scenario for API endpoints to avoid massive responses with large payloads. Usually, you don't need all the available data and it's sufficient to just fetch the subsets required. We've walked you through the setup of paginated requests using three different approaches: query parameters, response header with dynamic urls and range headers.

To recap your learnings, the following take aways are the keys of this chapter:

- [x] Proper response status codes for partial content
- [x] How to implement query parameter based pagination
- [x] Use dynamic urls to fetch partial content
- [x] Apply range headers to paginate through a dataset

In this chapter you've learned how to implement a common and important use case using Retrofit: pagination. In the next few chapters we'll focus on how to implement cleaner and better code, as well as boosting your productivity. The next chapter will show you ways to handle and parse errors and how to display user-friendly error messages.

# Chapter 7 — Error Handling

When doing networking on mobile devices, there is a lot that can go wrong. The request itself can get lost or dropped on a weak connection. Furthermore, the network part can go perfect, but the request itself is incorrect and the server responds with an error. In this chapter, we'll go through the various error scenarios. You'll learn how to take care of failure situations.

Most of the time, you need to manually apply the correct action like showing an error message as user feedback. If you get more than just the response status code, you can use the additional data to set the user in the right context and provide more information about the current error situation.

## Error Handling Preparations

Even though you want your app to always work like expected and there shouldn't be any issues while executing requests. However, you're not in control of when servers will fail or users will put wrong data which results in errors returned from the requested API. In those cases, you want to provide as much feedback to the user as required to set him/her into the right context so that he/she understands what the issue is.

Before diving into the actual request which results in an error, we're going to prepare classes to parse the response body which contains more information.

### Error Object

At first, we create the error object representing the response you're receiving from your requested API. Let's assume your API sends a JSON error body like this:

```
{
    statusCode: 409,
    message: "Email address already registered"
}
```

If we would just show the user a generic error message like `There went something wrong`, he/she would immediately be upset about this foolish app which isn't able to show what went wrong.

To avoid these bad user experiences, we're mapping the response body to a Java object, represented by the following class.

```java
public class APIError {

    private int statusCode;
    private String message;

    public APIError() {
    }

    public int status() {
        return statusCode;
    }

    public String message() {
        return message;
    }
}
```

We don’t actually need the status code inside the response body, it’s just for illustration purposes and this way you don’t need to extra fetch it from the response.

## Simple Error Handler

We’ll make use of the following class only having one `static` method which returns an `APIError` object. The `parseError` method expects the response as parameter. Further, you need to make your Retrofit instance available to apply the appropriate response converter for the received JSON error response.

```java
public class ErrorUtils {

    public static APIError parseError(Response<?> response) {
        Converter<ResponseBody, APIError> converter =
                ServiceGenerator.retrofit()
                        .responseBodyConverter(
                            APIError.class,
                            new Annotation[0]);

        APIError error;

        try {
            error = converter.convert(response.errorBody());
        } catch (IOException e) {
            return new APIError();
        }

        return error;
    }
}
```

We’re exposing our Retrofit instance from `ServiceGenerator` via a static method `ServiceGenerator.retrofit()`:

```
public class ServiceGenerator {

    private static Retrofit retrofit;

    public static Retrofit retrofit() {
        return retrofit;
    }

    // other fields & methods
    // see their respective chapters or our example project
    // ...
}
```

Please bear with us that we’re using a kind of hacky style by exposing the Retrofit object via static method. The thing that is required to parse the JSON error is the response converter. The response converter is available via our Retrofit object.

At first, we’re getting the error converter from the `ServiceGenerator.retrofit()` instance by additionally passing our `APIError` class as the parameter to the `responseBodyConverter` method. The `responseConverter` method will return the appropriate converter to parse the response body type. In our case, we’re expecting a JSON converter, because we’ve received JSON data. As you know, Retrofit is smart and selects the correct converter automatically. Consequently, if you’ve configured a JSON and XML converter, Retrofit will return the appropriate converter.

Further, we call `converter.convert` to parse the received response body data into an `APIError` object. Afterwards, we’ll return the created object.

## Error Handler in Action

Retrofit 2 has a different concept of handling “successful” requests than Retrofit 1. In Retrofit 2, all requests that can be executed (sent to the API) and for which you’re receiving a response are seen as “successful”. That means, the `onResponse` callback is fired for these requests and you need to manually check whether the request is actually successful (status 200-299) or erroneous (status 400-599).

If the request finished successfully, we can use the response object and do whatever we wanted. In case the error actually failed (remember, status 400-599), we want to show the user appropriate information about the issue.

```
Call<User> call = service.me();
call.enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        if (response.isSuccess()) {
            // use response data and do some fancy stuff :)
        } else {
            // parse the response body …
            APIError error = ErrorUtils.parseError(response);
            // … and use it to show error information

            // … or just log the issue like we’re doing :)
            Log.d("error message", error.message());
        }
    }

    @Override
    public void onFailure(Call<User> call, Throwable t) {
        // there is more than just a failing request
        // like: no internet connection
    }
});
```

As you can see, we use the `ErrorUtils` class to parse the error body and get an `APIError` object. Use this object and the contained information to show a meaningful message instead of a generic error phrase.

As you’ve seen in the beginning of this book, the `onFailure` callback method gets executed when the network request went fundamentally wrong, for example due to a missing Internet connection.

## Chapter Summary

This article shows you a simple way to manage errors and extract information from the response body. Most APIs will send you specific information on what went wrong and you should make use of it. This is just the tip of the iceberg when it comes to error handling. However, it’s a very app-

and domain-specific topic. On a high level, this chapter should give you the tools to deal with errors gracefully.

Let’s review what we’ve worked through in this chapter:

- [x] What different errors can occur with network requests
- [x] How Retrofit differentiates between network failures and server error codes
- [x] How to access and parse error responses

Excellent error handling is an important part of developing an app. But unexpected errors can and will already happen during your development cycle. If you need to drill down on issues, logging the requests and responses can be very helpful. The next chapter introduces you to logging in Retrofit.

# Chapter 8 — Logging

After diving deep into the art of using Retrofit for all kinds of network requests, in this chapter we'll try to improve your own productivity . Every developer needs to know his tools to achieve results fast and without too much struggle.

This might be a chapter which a lot of readers are tempted to skip, but we highly recommend a throughout read. It's important that you understand how to log what Retrofit is doing under the hood. It'll highly increase your chances to track down bugs in a few minutes rather than a few hours.

## Log Requests and Responses with Retrofit

Retrofit `1` integrated a logging feature for basic request and response debugging. The logging functionality was removed in Retrofit `2`, because the logs never reflected what actually was sent and just a guess on Retrofit's side. Also, the required HTTP layer is now completely based on OkHttp. Since many developers asked for logging capabilities in Retrofit `2`, the developers of OkHttp added a logging interceptor in the `2.6.0` release. This section will show you how to add and use the logging interceptor in your Android app in combination with Retrofit.

Retrofit `2` completely relies on OkHttp for any network operation. Due to the fact that OkHttp is a peer dependency of Retrofit, you won't need to add an additional dependency. If you are importing a specific OkHttp version older than `2.6.0`, you'll need to manually import the logging intercepter[42]. Add the following line to your gradle imports within your `build.gradle` file to fetch the logging interceptor dependency:

```
# only necessary for Retrofit 2 users,
# who import a specific OkHttp version older than 2.6.0
compile 'com.squareup.okhttp3:logging-interceptor:3.2.0'
```

### Add Logging to Retrofit 2

While developing your app and for debugging purposes it's nice to have a log feature integrated to show request and response information. Because logging isn't integrated by default anymore in Retrofit, we need to add a logging interceptor for OkHttp. Luckily, OkHttp already ships with this interceptor and you only need to activate it for your OkHttpClient.

[42]https://github.com/square/okhttp/tree/master/okhttp-logging-interceptor

```java
HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
// set your desired log level
logging.setLevel(Level.BODY);

OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
// add your other interceptors …

// add logging as last interceptor
httpClient.addInterceptor(logging);  // <-- this is the important line!

Retrofit retrofit = new Retrofit.Builder()
   .baseUrl(API_BASE_URL)
   .addConverterFactory(GsonConverterFactory.create())
   .client(httpClient.build())
   .build();
```

We recommend to add logging as the last interceptor, because this will also log the information which you added or manipulated with previous interceptors to your request.

## Log Levels

As you've seen in the code snippet above, we're calling `logging.setLevel(Level.BODY);` to configure how much of the requests and responses is getting logged. Logging too much information will blow up your Android monitor, too few information might not give you everything you need. That's why OkHttp's logging interceptor has four log levels: `NONE`, `BASIC`, `HEADERS`, `BODY`. We'll walk you through each of the log levels and describe their output.

### None

**No logging.**

Use this log level for production environments to enhance your apps performance by skipping any logging operation.

### Basic

**Log request type, url, size of request body, response status and size of response body.**

```
D/HttpLoggingInterceptor$Logger: --> POST /upload HTTP/1.1 (277-byte body)
D/HttpLoggingInterceptor$Logger: <-- HTTP/1.1 200 OK (543ms, -1-byte body)
```

Using the log level `BASIC` will only log minimal information about your request. If you're just interested in the request body size, response body size and response status, this log level is the right one.

## Headers

**Log request and response headers, request type, url, response status.**

Using the `HEADERS` log level will only log request and response headers. Retrofit or OkHttp will add appropriate request headers by default, but they won’t show up on your request since they are added later in the request chain. If you didn’t intercept the actual request on Android, there is nothing to see. If you add request headers yourself, make sure the logging interceptor is the last interceptor added to the OkHttp client. If you add the interceptor first, there isn’t any header data set on the request yet.

We use the two header fields `Accept` and `Content-Type` to illustrate the output if you define the values yourself.

```
D/HttpLoggingInterceptor$Logger: --> POST /upload HTTP/1.1
D/HttpLoggingInterceptor$Logger: Accept: application/json
D/HttpLoggingInterceptor$Logger: Content-Type: application/json
D/HttpLoggingInterceptor$Logger: --> END POST
D/HttpLoggingInterceptor$Logger: <-- HTTP/1.1 200 OK (1039ms)
D/HttpLoggingInterceptor$Logger: content-type: text/html; charset=utf-8
D/HttpLoggingInterceptor$Logger: cache-control: no-cache
D/HttpLoggingInterceptor$Logger: vary: accept-encoding
D/HttpLoggingInterceptor$Logger: Date: Wed, 28 Oct 2015 08:24:20 GMT
D/HttpLoggingInterceptor$Logger: Connection: keep-alive
D/HttpLoggingInterceptor$Logger: Transfer-Encoding: chunked
D/HttpLoggingInterceptor$Logger: OkHttp-Selected-Protocol: http/1.1
D/HttpLoggingInterceptor$Logger: OkHttp-Sent-Millis: 1446020610352
D/HttpLoggingInterceptor$Logger: OkHttp-Received-Millis: 1446020610369
D/HttpLoggingInterceptor$Logger: <-- END HTTP
```

Besides the headers of server’s response, you’ll get the information which protocol was selected and the respective milliseconds when your request was sent and the response was received.

## Body

**Log request and response headers and body.**

This is the most complete log level and will print out every related information for your request and response.

```
D/HttpLoggingInterceptor$Logger: --> POST /upload HTTP/1.1
D/HttpLoggingInterceptor$Logger: --9df820bb-bc7e-4a93-bb67-5f28f4140795
D/HttpLoggingInterceptor$Logger: Content-Disposition: form-data; name="descripti\
on"
D/HttpLoggingInterceptor$Logger: Content-Transfer-Encoding: binary
D/HttpLoggingInterceptor$Logger: Content-Type: application/json; charset=UTF-8
D/HttpLoggingInterceptor$Logger: Content-Length: 37
D/HttpLoggingInterceptor$Logger:
D/HttpLoggingInterceptor$Logger: "hello, this is description speaking"
D/HttpLoggingInterceptor$Logger: --9df820bb-bc7e-4a93-bb67-5f28f4140795--
D/HttpLoggingInterceptor$Logger: --> END POST (277-byte body)
D/HttpLoggingInterceptor$Logger: <-- HTTP/1.1 200 OK (1099ms)
D/HttpLoggingInterceptor$Logger: content-type: text/html; charset=utf-8
D/HttpLoggingInterceptor$Logger: cache-control: no-cache
D/HttpLoggingInterceptor$Logger: vary: accept-encoding
D/HttpLoggingInterceptor$Logger: Date: Wed, 28 Oct 2015 08:33:40 GMT
D/HttpLoggingInterceptor$Logger: Connection: keep-alive
D/HttpLoggingInterceptor$Logger: Transfer-Encoding: chunked
D/HttpLoggingInterceptor$Logger: OkHttp-Selected-Protocol: http/1.1
D/HttpLoggingInterceptor$Logger: OkHttp-Sent-Millis: 1446021170095
D/HttpLoggingInterceptor$Logger: OkHttp-Received-Millis: 1446021170107
D/HttpLoggingInterceptor$Logger: Perfect!
D/HttpLoggingInterceptor$Logger: <-- END HTTP (8-byte body)
```

This is the only log level where you'll get the response body data. If you're in an argument with your backend developer, use this log level to show the received response data. However, the `BODY` log level will clutter your Android monitor if you're receiving large data sets. Use this level only if necessary.

We hope you can apply logging to your development flow and the shown information about the log level will help you choose the right one for you. Use different log levels and try what works best for you. Even if the `BODY` log level might clutter up your log, maybe you'll find your desired information only showing every request detail. Nevertheless, we recommend to only activate the logging in development builds. Don't add the logging interceptor in your app's production release APK!

## Enabling Logging for Debug Builds Only

Automation is one of the best tools to increase developer focus and productivity. Enabling and disabling logging for Retrofit can be one of the tedious, repetitive tasks which steal your attention. Furthermore, it increases the chance that the logging is still enabled for a production build. So let's automate this process: logging will be enabled for debug builds during your development process; and logging will be disabled for all production versions of your app!

The solution is pretty simple: we'll utilize the `BuildConfig.DEBUG` boolean variable which is provided by the Android framework. It will return `true` for your development builds and `false` for your production builds.

A little earlier in this chapter we've shown you this code snippet:

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

OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
httpClient.addInterceptor(logging);

Retrofit retrofit = new Retrofit.Builder()
   .baseUrl(API_BASE_URL)
   .addConverterFactory(GsonConverterFactory.create())
   .client(httpClient.build())
   .build();
```

It's time to slightly change the code to only enable logging for debug builds:

```
OkHttpClient.Builder httpClient = new OkHttpClient.Builder();

if (BuildConfig.DEBUG) {
    HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
    logging.setLevel(Level.BODY);

    httpClient.addInterceptor(logging);
}

Retrofit retrofit = new Retrofit.Builder()
   .baseUrl(API_BASE_URL)
   .addConverterFactory(GsonConverterFactory.create())
   .client(httpClient.build())
   .build();
```

This will only add the logging intercepter if it's a development build. We highly recommend applying this or a similar approach to your app. It's not going to be hours, but it's going to save you a little time every day. Your time matters, so let's wrap up this chapter.

## Chapter Summary

In this chapter you've learned how to log requests and responses with Retrofit. Make sure you understood all the important take-aways:

- [x] How to integrate OkHttp Logging Interceptor
- [x] Understand the various logging levels
- [x] How to enable logging for development builds only
- [x] Don’t use logging in production builds

Speaking of production builds, the next section will show you how to support multiple server environments and how to quickly change Retrofit’s base URL during runtime. This makes it easy to test your app against multiple environments in a matter of minutes without having to compile a new APK every time.

# Chapter 9 — Multiple Server Environments

When you're developing an app professionally, you're probably dealing with multiple server environments. Your API developers are constantly changing and improving the endpoints the apps consume. If you've an excellent backend developer, he''ll provide multiple environments:

- **develop**: latest snapshot of the API code
- **staging**: next API iteration up for a deployment onto the production environment
- **production**: API version all users of the public release are using

The details might look different in your case, but the general concept should still stand. In this chapter, we'll show you how to make it easy for you to build & test against the various environments.

Additionally, the second section will show you how your app can change the base URL at runtime. This also requires a little setup work, but will save you a lot of time in the long-term future.

## Product-Flavor-Dependent Base URLs

This section is for you if your project has multiple server environments and your app code currently looks like this:

```
public static String apiBaseUrl = "https://futurestud.io/api";
//public static String apiBaseUrl = "https://staging.futurestud.io/api";
//public static String apiBaseUrl = "https://dev.futurestud.io/api";
```

Don't work with comments to change the base URL to switch server environment! It'll just cause headaches in the future and easily leads to simple mistakes.

### Setup of Product-Flavors

We recommend to setup multiple Gradle product flavors. If you haven't done that already, edit your `build.gradle` and add:

```
android {
    productFlavors {
        dev {
            applicationId "io.futurestudio.awesomeapp.dev"
            versionCode 1
            versionName "1.0.0"
        }

        staging {
            applicationId "io.futurestudio.awesomeapp.staging"
            versionCode 1
            versionName "1.0.0"
        }

        production {
            applicationId "io.futurestudio.awesomeapp"
            versionCode 1
            versionName "1.0.0"
        }
    }

    // other configs
    // ...
}
```

This will give you the option to build a different APK for each environment. Since we've configured a different `applicationId` for each product flavor, you can have all three apps installed at the same time. It makes it super easy to test the app against each server environment. The `versionCode` and `versionName` are just for demonstration purposes and are irrelevant regarding the base URL.

## Setup of Product-Flavor-Dependent Base URL

The next step is to change the base URL for each product flavor to the appropriate server environment. Currently, the `ServiceGenerator` looks like this:

```
public class ServiceGenerator {

    public static String apiBaseUrl = "https://futurestud.io/api";
    //public static String apiBaseUrl = "https://staging.futurestud.io/api";
    //public static String apiBaseUrl = "https://dev.futurestud.io/api";

    private ServiceGenerator() {
    }

    // other fields & methods
    // ...
}
```

The trick is instead of hard-coding a String into the code, you're moving it into an Android string resource. Since you need a `Context` to access String resources, you cannot set it directly to a static field. We need to extend the `ServiceGenerator` constructor and set the `apiBaseUrl` field when it's not set yet:

```
public class ServiceGenerator {
    public static String apiBaseUrl;

    private ServiceGenerator(Context context) {
        if (TextUtils.isEmpty(apiBaseUrl)) {
            apiBaseUrl = context.getString(R.string.server_base_url)
        }
    }

    // other fields & methods
    // ...
}
```

Finally, open Finder or Windows Explorer and create a resource of `R.string.server_base_url` for each product flavor:

**src/dev/res/values/server_base_url.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="server_base_url">https://dev.futurestud.io/api</string>
</resources>
```

**src/staging/res/values/server_base_url.xml**:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="server_base_url">https://staging.futurestud.io/api</string>
</resources>
```

**src/production/res/values/server_base_url.xml**:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="server_base_url">https://futurestud.io/api</string>
</resources>
```

You can now build an APK for each server environment and the `ServiceGenerator` will automatically select the correct base URL. This approach is very helpful if you need to strictly differentiate between the various environments. For example, it helps a lot if you've to support an entire alpha/beta/production app chain.

If you literally just want to use it to test your app on your own against the various environments, the next section might be something for you.

## Change API Base URL at Runtime

In this section we'll show you a very handy tool, which can make your debugging life so much easier. If you're an Android app developer and you've to deal with multiple versions of an API (for example develop, staging and production) you're probably tired of building three (or more) versions of the app, if you just quickly want to see if your app works against all server deployments.

In the next few minutes, you'll learn how you can change the API base url at runtime! This means, you can check with one compilation and installation how your app behaves with all API versions.

### The Core: Extending the ServiceGenerator

You've read a lot about the `ServiceGenerator` class in the first few chapters. In the current version the `ServiceGenerator` works with multiple static fields and a String constant `API_BASE_URL`, which holds the API base url:

```java
public class ServiceGenerator {
        public static final String API_BASE_URL = "https://futurestud.io/api";

        private ServiceGenerator() {

        }

        // other fields & methods
        // ...
}
```

## Adjusting the ServiceGenerator

With this setup you don't have a chance to change the `API_BASE_URL` constant at runtime. You've to change it in the source code, compile a new `.apk` and test it again. Because this is very inconvenient if you're working with multiple API deployments, we'll make minor changes to the `ServiceGenerator` class:

```java
public class ServiceGenerator {
    public static String apiBaseUrl = "http://futurestud.io/api";

    private ServiceGenerator() {

    }

    public static void changeApiBaseUrl(String newApiBaseUrl) {
        apiBaseUrl = newApiBaseUrl;

        builder = new Retrofit.Builder()
                        .addConverterFactory(GsonConverterFactory.create())
                        .baseUrl(apiBaseUrl);
    }

    // other fields & methods
    // ...
}
```

Let's examine our changes. We've renamed the constant `API_BASE_URL` to a non-final field `apiBaseUrl`. We also added a new static method `changeApiBaseUrl(String newApiBaseUrl)`, which will change that particular `apiBaseUrl` variable. It also creates a new version of the `Retrofit.Builder` instance `builder`. This is important because we're re-using the `builder` for requests. If we don't create a new instance all requests still would have gone against the original `apiBaseUrl` value.

## Example Usage

We've made the necessary changes to the `ServiceGenerator`. Let's move on to actually utilize our new functionality:

```java
public class DynamicBaseUrlActivity extends AppCompatActivity {

    public static final String TAG = "CallInstances";
    private Callback<ResponseBody> downloadCallback;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_file_upload);

        downloadCallback = new Callback<ResponseBody>() {
            @Override
            public void onResponse(
                    Call<ResponseBody> call, Response<ResponseBody> response) {
                Log.d(TAG, "server contacted at: " + call.request().url());
            }

            @Override
            public void onFailure(Call<ResponseBody> call, Throwable t) {
                Log.d(TAG, "call failed. url: " + call.request().url());
            }
        };

        // first request
        FileDownloadService downloadService =
            ServiceGenerator.create(FileDownloadService.class);
        Call<ResponseBody> originalCall =
            downloadService.downloadFileWithFixedUrl();
        originalCall.enqueue(downloadCallback);

        // change base url
        ServiceGenerator
            .changeApiBaseUrl("https://development.futurestud.io/api");

        // new request against new base url
        FileDownloadService newDownloadService =
            ServiceGenerator.create(FileDownloadService.class);
        Call<ResponseBody> newCall =
            newDownloadService.downloadFileWithFixedUrl();
        newCall.enqueue(downloadCallback);
    }
}
```

In this activity we're showing you two requests to download a file from the server. If you're interested in the details on how to download files with Retrofit, head over to the fifth chapter. After the first request is executed, we change the base url to our development environment with the new `ServiceGenerator.changeApiBaseUrl()` method. Lastly, we'll make the same download request again. When we start the app, we're getting the following logs:

```
D/CallInstances: server contacted at: http://futurestud.io/resource/example.zip
D/CallInstances: server contacted at: http://development.futurestud.io/resource/\
example.zip
```

This is exactly as we wanted Retrofit to behave. The first request still goes against the production server. The second request, after changing the base url, goes against our development server. Awesome!

## When To Change the Base Url at Runtime

The demo code above simplifies things a little. We usually implement a button in the debug version of the app, where the tester can select the wished server environment. Thus, depending on your situation, you probably have to write a bit more code to decide when and to which base url Retrofit should change to.

Also, we only recommend doing this for debugging purposes. We don't think this is a good way of making your app work with various servers at the same time. If your app needs to deal with more than one API, look for a different version. The dynamic url section in the second chapter might be a good start.

Lastly, please test if you can simply switch environments with your app. For example, if you store user and authentication information on the server side, switching the environment might cause problems. Your production database most likely does not contain the same users as your development database, correct? In our apps, we delete all relevant user data and force a new, fresh login after the tester changed the environment via a new base url.

## Section Summary

In this section, you've learned how to change the API base url at runtime. This can be incredibly useful if you're dealing with multiple API deployments. We've shown you the necessary enhancement to the `ServiceGenerator` class and how to make the required requests. You also have to be aware of the possible consequences when switching API deployments during runtime. Nevertheless, if you spent the hour to implement this for your app, you'll save days of compiling time!

## Chapter Summary

Hopefully this chapter was as valuable to you as it was to us when we implemented the two strategies for the first time. You should have learned two new things. First, you've learned how you can set up multiple app product flavors to reflect each server environment. This is awesome if your project goes through an alpha/beta/release cycle. Second, you've seen a simpler case where you can quickly change the base URL at runtime. This can be very helpful if you need to test your app against the various API versions.

Before moving to the next chapter, you might want to double check all the key take-aways:

- [x] Why should you setup multiple app product flavors
- [x] How to setup multiple product flavors
- [x] How to make the base URL product-flavor-dependent
- [x] How to change the base URL at runtime
- [x] What pitfalls can occur when you change the base URL at runtime

In the next chapter we'll show you how you can mock server responses from the client-side! It'll also include an introduction to simulating various network behaviors.

# Chapter 10 — Client-Side Mocking of Server Responses

In the past chapters we've shown you a wide variety of Retrofit features. In this chapter we'll explore a new area: mocking. Specifically, we'll introduce you techniques to set up mock server responses on the phone.

## Why Mocking Responses Make Sense

Developing your app against a deployed backend requires you to always have an Internet connection. Additionally, the backend functionality must provide all the features you're currently implementing on the client side. Let's assume a typical situation: you have an appointment somewhere outside your daily work place and want to fill the boring commute. At least in Germany, the Internet connection during train rides is quite unstable. While that is a great test for your app, you'll be wasting more time waiting for responses than actually working on your app.

This is your chance to create a mock client and fake API responses locally. Of course, this approach has its pros and cons. An advantage is that no Internet connection is required, because there won't be any requests sent to the API. A disadvantage is that you have to implement at least the mock responses on the client side and perhaps you duplicate some backend logic, too.

However, mock clients are a great way to decouple your development from the backend in case you know the representation of responses and can fake it until the backend makes it.

## Create Your Own MockRetrofit Implementation

You have multiple options to mock your API requests with Retrofit. In this section, we'll explore the `MockRetrofit` library of the Retrofit creators. It offers an easy way to mock server responses and emulate various aspects of network communication. Before we start, we need to add the library as a dependency to our `build.gradle`:

```
// NEW: mocking responses
compile 'com.squareup.retrofit2:retrofit-mock:2.0.0'

// ...
// other dependencies, like:
// Retrofit
compile 'com.squareup.retrofit2:retrofit:2.0.0'

// Retrofit Standard Converters
compile 'com.squareup.retrofit2:converter-gson:2.0.0'
```

The great thing about `MockRetrofit` is that you can leave a lot of your configuration and re-use your service interfaces. In order to make this guide a little more visual, we'll use the same interface as within the GitHub pagination chapter. Thus, our activity is going to need a `ListView` for displaying a list of GitHub repositories of a user. We're also (partially) re-creating the following interface:

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

The `GitHubRepository` class is a very simple model for a Github repository:

```java
public class GitHubRepository {

    private int id;
    private String name;

    public GitHubRepository() {
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

We can leave our Retrofit instance creation the same:

```java
// setup standard retrofit
Retrofit retrofit =
    new Retrofit.Builder()
        .baseUrl("https://api.github.com/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
```

The next thing we'll need is a class to emulate the network behavior:

```java
// set up emulated network behavior
NetworkBehavior networkBehavior = NetworkBehavior.create();
```

The `NetworkBehavior` class offers a variety of options to emulate real world situations. Later in this chapter, we'll dive deeper into this topic. For now it's sufficient to accept the default error, standard values of a 2 second mean respond time and a failure rate of 3%.

The next step is to wrap the Retrofit instance from the real GitHub API into our mock environment. Here we can finally utilize the `MockRetrofit` class:

```java
MockRetrofit mockRetrofit =
    new MockRetrofit.Builder(retrofit)
        .networkBehavior(networkBehavior)
        .build();
```

When creating the `MockRetrofit` object we need to pass our network behavior configuration, which was left in its default settings.

After we've set up our `MockRetrofit` object, we can put it to work and create the service for the GitHub API interface. As mentioned before, the API interface declaration is identical as in the regular Retrofit usage. Nevertheless, the return type is a bit different:

```java
BehaviorDelegate<GitHubService> githubBehaviorDelegate =
        mockRetrofit.create(GitHubService.class);
```

As you can see, creating a service with `MockRetrofit` doesn't return the service directly. It wraps it in a `BehaviorDelegate` class. We'll need the created delegate object for our final step of the implementation: creating and returning mock responses!

## Mocking Responses From a Service Interface

In order to mock server behavior, we need to create a new class, which implements our API interface class. Remember, we're mocking the `GitHubService` interface, so we're writing an implementation for it:

```
class MockGitHub implements GitHubService {

    private final BehaviorDelegate<GitHubService> githubBehaviorDelegate;

    public MockGitHub(BehaviorDelegate<GitHubService> delegate) {
        this.githubBehaviorDelegate = delegate;
    }
}
```

Our created `MockGitHub` class gets the `BehaviorDelegate` object passed in its constructor. We'll use it to return our (mocked) responses. But before we can do that, we have to create some mock data! Luckily, we can directly work with Java objects. In order to make the functionality closer to the actual API we'll create a `Map<User, List of Repositories>`, which maps a GitHub user name to a list of repositories. We store the mocked data in a global variable, after it has been created:

```
class MockGitHub implements GitHubService {

    private final BehaviorDelegate<GitHubService> githubBehaviorDelegate;
    private final Map<String, List<GitHubRepository>> ownerRepoMap;

    public MockGitHub(BehaviorDelegate<GitHubService> delegate) {
        this.githubBehaviorDelegate = delegate;

        ownerRepoMap = new LinkedHashMap<>();

        // we're filling the local list with some data
        addFakeGithubRepo("square", "retrofit");
        addFakeGithubRepo("fs-opensource", "strider");
        addFakeGithubRepo("fs-opensource", "android-tutorials-customfont");
        addFakeGithubRepo("fs-opensource", "hapi-rethinkdb-dash");
    }

    public void addFakeGithubRepo(String owner, String repo) {
        List<GitHubRepository> reposForOwner = ownerRepoMap.get(owner);

        if (reposForOwner == null) {
            reposForOwner = new ArrayList<>();
            ownerRepoMap.put(owner, reposForOwner);
        }

        GitHubRepository githubRepo = new GitHubRepository();
        githubRepo.setName(repo);
        reposForOwner.add(githubRepo);
    }
}
```

Now we've a list of Github repositories sorted by owner in the `ownerRepoMap` variable. The final step is to bind the API endpoints to that list and actually only return correct data:

```
class MockGitHub implements GitHubService {

    private final BehaviorDelegate<GitHubService> githubBehaviorDelegate;
    private final Map<String, List<GitHubRepository>> ownerRepoMap;

    public MockGitHub(BehaviorDelegate<GitHubService> delegate) {
        this.githubBehaviorDelegate = delegate;

        ownerRepoMap = new LinkedHashMap<>();

        // we're filling the local list with some data
        addFakeGithubRepo("square", "retrofit");
        addFakeGithubRepo("fs-opensource", "strider");
        addFakeGithubRepo("fs-opensource", "android-tutorials-customfont");
        addFakeGithubRepo("fs-opensource", "hapi-rethinkdb-dash");
    }

    public void addFakeGithubRepo(String owner, String repo) {
        List<GitHubRepository> reposForOwner = ownerRepoMap.get(owner);

        if (reposForOwner == null) {
            reposForOwner = new ArrayList<>();
            ownerRepoMap.put(owner, reposForOwner);
        }

        GitHubRepository githubRepo = new GitHubRepository();
        githubRepo.setName(repo);
        reposForOwner.add(githubRepo);
    }

    // NEW:
    @Override
    public Call<List<GitHubRepository>> reposForUser(@Path("user") String user) {
        List<GitHubRepository> response = ownerRepoMap.get(user);

        return githubBehaviorDelegate
                .returningResponse(response)
                .reposForUser(user);
    }
}
```

The `reposForUser` endpoint returns the list of repositories for a specific user. Since our mock data is already prepared for this scenario we only need to access it via `ownerRepoMap.get(user);`. The final step is to return it with the `githubBehaviorDelegate` object:

```
return githubBehaviorDelegate
    .returningResponse(response)
    .reposForUser(user);
```

And this is all you need to mock Server responses within an Android app! Of course, we haven't made the call to the fake API endpoint yet, but this is exactly the same as a regular Retrofit call:

```
// make the call, just like a regular one
Call<List<GitHubRepository>> call = mockGitHub.reposForUser("fs-opensource");
call.enqueue(new Callback<List<GitHubRepository>>() {
    @Override
    public void onResponse(Call<List<GitHubRepository>> call,
                           Response<List<GitHubRepository>> response) {
        displayGithubRepositories(listView, response.body());
    }

    @Override
    public void onFailure(Call<List<GitHubRepository>> call, Throwable t) {
        displayGithubRepositories(listView, new ArrayList<GitHubRepository>());
    }
});
```

In this section you've seen the basics of mocking responses with `MockRetrofit`. It takes a while to understand the concept, but it's well designed and minimizes the amount of copy & pasting code. We love that it's easy to switch between the real world API and the mocked API responses. In the next section we're taking a closer look at the `NetworkBehavior` options.

## Customizing Network Behavior of Mocked Server Responses

In our last section, we've introduced mocking server responses with `MockRetrofit`. We've mentioned on the side that we're using the NetworkBehavior[43] class, but didn't go into the details of the

[43]https://github.com/square/retrofit/blob/master/retrofit-mock/src/main/java/retrofit2/mock/NetworkBehavior.java

configuration options. We'll make up for that now.

## Overview of Network Behavior Options

The `NetworkBehavior` class gives us four configuration options:

- `networkBehavior.setDelay();`
- `networkBehavior.setVariancePercent();`
- `networkBehavior.setFailurePercent();`
- `networkBehavior.setFailureException();`

We'll go through each option the methods represent in the next few sections.

### Delay

**Default Response Delay: 2 Seconds**

The first setting is changing how long the mocked server will take to respond to your request. By default, it's set to 2 seconds. You can change that value by calling `.setDelay(value, unit)`, for example:

```
networkBehavior.setDelay(15, TimeUnit.SECONDS);
```

This would change the average response time to 15 seconds. The second parameter is a TimeUnit[44] constant.

It's important to understand that the `NetworkBehavior` class has some randomization build in. This means for you that the response time isn't always exactly 2 seconds (or for whatever you set it to). This option can be very valuable if you want to test how your app's UI behaves with long running requests.

If you don't like the randomization, the next setting is for you…

### Variance Percent

**Default Variance: 40%**

The response time based on the default is randomized a bit. The `NetworkBehavior` class has a `setVariancePercent()` method to adjust the response time variance. The method accepts a range of 0 to 100 percent. By default, it's set to 40%.

If you don't want any variety in your response time and always get the exact same delay, set it to 0:

[44]http://developer.android.com/reference/java/util/concurrent/TimeUnit.html

```
networkBehavior.setVariancePercent(0);
```

If you want to increase the delay range (meaning the change from the base value increases, so even shorter values are possible), set it to a higher value, for example 60:

```
networkBehavior.setVariancePercent(60);
```

### Failure Percent

**Default Failure Rate: 3%**

Especially when users are on a mobile Internet connection, requests will fail every once in a while. Your app needs to be able to deal with failed connections either by automatically retrying or offering the user the option to refresh the content. The `NetworkBehavior` implementation respects this challenge by having a fail rate build in. By default, 3% of your requests will be mocked as failed when you're using `MockRetrofit`. Of course, you can change that with `setFailurePercent()`, which accepts a value between 0 and 100 percent:

```
networkBehavior.setFailurePercent(0); // no request fails

networkBehavior.setFailurePercent(100); // every request fails
```

### Failure Exception

**Default Exception: IOException("Mock failure!");**

The request can fail due to a variety of reasons on the network level. Retrofit passes the exception to the `onFailure()` callback. If your app is reacting to different exceptions, you can mock these with the `.setFailureException()` method. You can specify what exactly failed in your mocked environment:

```
`networkBehavior.setFailureException(new UnknownHostException());`
```

In this section, you've learned the options to fine-tune the network settings when mocking API responses.

## Chapter Summary

We hope this chapter showed you the basics of mocking server responses with `MockRetrofit`. The setup is fairly simple and can be quite helpful in a variety of situations. Make sure you understood all key concepts before moving on:

- [x] What is `MockRetrofit`

- [x] Why and when should we mock server responses
- [x] Which network behavior options are available

With this chapter we're concluding the Retrofit development chapters. The next chapter describes the configuration of ProGuard when using Retrofit within your app. Further, we give you a complete ProGuard configuration to copy & paste into your project. ProGuard can be an annoying topic shortly before your app release. Verify that your app still works with ProGuard enabled in advance to your Google Play distribution.

---

Resources:

- Retrofit 2 Mocking Tutorial[45]
- Official Example of MockRetrofit[46]

[45]http://riggaroo.co.za/retrofit-2-mocking-http-responses/
[46]https://github.com/square/retrofit/blob/master/samples/src/main/java/com/example/retrofit/SimpleMockService.java

# Chapter 11 — App Release Preparation

Google highly recommends to use ProGuard for release APKs that get distributed via Google Play. We'll guide you through the basic setup of ProGuard and show you how to configure Retrofit in the phase of release preparation.

First, let's shortly recap ProGuard and afterwards we dive into exemplary ProGuard rules to keep your app working after compiling it with ProGuard enabled.

## Enable ProGuard

First things first: What is ProGuard[47]? ProGuard is a tool integrated with the Android build system that shrinks, optimizes and obfuscates your app code by deleting unnecessary code and renaming classes, methods and fields with cryptically names.

That sounds like there is some magic going on in the background when activating ProGuard. Actually, the result of all this is a smaller sized APK file that is much more difficult the reverse engineer. ProGuard only runs when you build your app in release mode and set the option `minifyEnabled` within your `build.gradle` file to `true`. The following code is an extract from ab app's `build.gradle`.

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                    'proguard-rules.pro'
        }
    }
}
```

The `minifyEnabled true` method activates ProGuard within the release build type. Additionally, the `proguardFiles getDefaultProguardFile('proguard-android.txt')` fetches the default ProGuard settings from Android's SDK located within the `tools/proguard` folder. The `proguard-rules.pro` file is located within your `app` folder and gets applied during the build phase as well.

[47] https://developer.android.com/tools/help/proguard.html

## Configure ProGuard Rules for Retrofit

ProGuard is very harsh when it comes to deleting code. In many situations, ProGuard doesn't analyzes the code correctly and removes classes which seem to be unused, but are actually needed in your app.

Getting a `ClassNotFoundException` within your app after running ProGuard is the indicator that you need to configure ProGuard to keep the class(es). The general syntax to say the ProGuard tool should keep a class:

```
-keep class <MyClass>
```

We use the following snippet within our apps that are distributed via Google Play. Because Retrofit doesn't integrate any converter by default anymore, we don't need to keep track of it directly. It'll be covered by the `retrofit2.**` package and don't get deleted using ProGuard. Copy the snippet to your `proguard-rules.pro` file to keep necessary Retrofit classes within your app.

```
# Retrofit
-dontwarn retrofit2.**
-keep class retrofit2.** { *; }
-keepattributes Signature
-keepattributes Exceptions

-dontwarn okhttp3.*
-dontwarn rx.**

-keep class javax.inject.** { *; }
-keep class javax.xml.stream.** { *; }

# Add any classes the interact with gson
# the following line is for illustration purposes
-keep class io.futurestud.retrofitbook.android.services.models.**
```

The ProGuard configuration above may not fit completely for your app. Different apps may require different ProGuard configurations to work properly when built in release mode. As already mentioned, watch out for `ClassNotFoundException` and add entries within the `proguard-rules.pro` file.

## Obfuscated Stack Traces

Whenever ProGuard runs, it obfuscates your classes, methods and fields and it's very hard to debug your code. The good thing: ProGuard outputs a `mapping.txt` file which maps the original class, method and field names with the obfuscated ones.

Use the `retrace.sh` on Mac and Linux or the `retrace.bat` on Windows to convert the obfuscated StackTrace into a comprehensible one. The `retrace.sh|bat` file is located within `<sdk_-root>/tools/proguard/` directory. The syntax to execute the `retrace` tool is this:

```
retrace.sh|bat [-verbose] mapping.txt [<stacktrace_file>]
```

Let's look at an actual command:

```
retrace.sh -verbose mapping.txt obfuscated_stacktrace.txt
```

**Keep your `mapping.txt` file for every release** you publish to users! Different app versions require different `mapping.txt` files to translate possible errors back to understandable class, method and field names.

## Chapter Summary

This chapter explains how to configure your Android project for the use of ProGuard with Retrofit. Keep in mind to save the `mapping.txt` files for stacktrace conversion and debugging. Error handling with obfuscated code is hell on earth if you lost the mapping file and need to find the issue within your code.

What you've learned within this chapter:

- [x] What is ProGuard
- [x] How to enable ProGuard
- [x] Define ProGuard rules that won't delete Retrofit related classes
- [x] The importance of the ProGuards mapping file to translate obfuscated code

It has been a lot of content about Retrofit. Your brain must be on fire (or not) and we hope you're more than before in love with Retrofit. If you're still with Retrofit 1.9 (or earlier), have a look at the appendix A. We've added a comprehensive upgrade guide for Retrofit 1.9 to 2.x.

# Outro

Our goal is to truly help you getting started and ultimately master Retrofit. We hope you learned many new things throughout this book. We want you to save time while learning the basics and details about Retrofit. The existing Retrofit documentation lacks various information and this book should help you to gain extensive in-depth knowledge without loosing time searching StackOverflow for correct answers.

We're currently planning new chapters and sections on Retrofit that will be added within the upcoming months. We feel the need for an extra chapter about reactive extensions on Android using RxAndroid/RxJava. That is the current high priority item on our idea list. Besides that, we plan to add content on the testing chapter.

Nonetheless, we'll update the content of this book to later Retrofit versions as new releases become available. However, it will take some time to update the code for breaking changes introduced to Retrofit. Of course, we'll let you about any book updates.

As a reader of this book, we'll always reward your loyalty by publishing exclusive content and any future update will —of course— be free for you!

For us it's really important to exceed our reader's expectations. In all our products and guides we aim for a high quality. If you feel like a section or chapter in this book wasn't clear or extensive enough, please let us know at info@futurestud.io[48]. We always love hearing back from you, so if you have anything to say, don't hesitate to shoot us an email. We welcome any feedback, critic, suggestions for new topics or whatever is currently on your Retrofit mind!

Don't forget, we're publishing new blog posts every Monday and Thursday, mainly about Android and Node.js within the Future Studio University. Feel free to visit our homepage[49] and the University[50] :)

Thanks a lot for reading this book! We truly appreciate your interest and hope you learned a lot from reading this book! <3

— Marcus & Norman

[48]mailto:info@futurestud.io
[49]https://futurestud.io
[50]https://futurestud.io/blog

# About the Book

This book is based on the Retrofit series[51] published in the Future Studio blog. Originally, the blog post series had 14 single posts, including a series round-up to provide a comprehensive overview of which topics are covered and what to expect within the articles.

We're overwhelmed about the amazing popularity of our Retrofit series! Thanks a lot for all of the positive feedback, comments and suggestions for additional articles! It's awesome to see a growing interest in the published posts. That's the reason why we've decided to write a book on Retrofit with additional content and more extensive code examples. You may ask yourself, "What are the differences between this book and the blog posts?" Fair enough, that's a good question to ask!

**You're getting the following benefits from buying this book**:

- **Additional content**: of course you'll profit from new content! Bro, we don't let you down. New chapters explain how to mock a REST client with Retrofit, how to use the OAuth refresh token to update your access token and how to configure ProGuard with Retrofit for Google Play releases.
- **Extensive code examples**: we've highly improved the code examples to provide more context where to put which annotations, classes, and so on.
- **Android Project included**: all code examples used within this book to provide more context of where to use them.
- **Nicely formatted and eReader ready**: you get `epup`, `mobi` and `PDF` files for download.

This book includes an Android project with code examples. We utterly recommend to download the code package from this book's extras and give it a spin. It contains all code examples used within this book and additionally sets you in the context of where to use them.

We are big proponents of fair use and value. Feel free to copy any code snippets from the book or the Android project into your apps. You don't have to mention or link back to us. We utterly value comments, emails and tweets if you benefit from the book or code. Those messages keep us motivated and everybody loves kind words of appreciation :)

- Follow us on Twitter: @futurestud_io[52]
- Email us: info@futurestud.io[53]
- Our blog provides valuable articles: futurestud.io/blog[54]

[51]https://futurestud.io/blog/retrofit-getting-started-and-android-client

[52]https://twitter.com/futurestud_io

[53]mailto:info@futurestud.io

[54]https://futurestud.io/blog

# Appendix — Retrofit 2 Upgrade Guide from 1.x

The previous chapters are all based on Retrofit 2 and is written for new apps and new developers starting with Retrofit. However, the development world is not always a green field project. If your app is currently implemented with Retrofit 1.9 and you need or want to upgrade to Retrofit 2.0, read on. This guide will help you push your app to the next version of Retrofit since there are multiple breaking changes when jumping to the upcoming version 2.

## Introduction

Retrofit 2 comes with various fundamental changes and also includes breaking changes to the internal API. That requires you to update your code related to Retrofit when jumping on Retrofit 2.

## Maven & Gradle Dependencies

Retrofit is available as Maven and Gradle dependencies. As within Retrofit 1, you need to import the underlying HTTP client. By default, Retrofit 2 leverages OkHttp for the job. That's why we also need to import the `okhttp` package.

**Gradle: Retrofit & OkHttp**

```
compile 'com.squareup.retrofit2:retrofit:2.0.0'
compile 'com.squareup.okhttp:okhttp:3.2.0'
```

**Maven: Retrofit & OkHttp**

```
<dependency>
  <groupId>com.squareup.retrofit2</groupId>
  <artifactId>retrofit</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency>
  <groupId>com.squareup.okhttp</groupId>
  <artifactId>okhttp</artifactId>
  <version>3.2.0</version>
</dependency>
```

Retrofit 2 doesn't ship with Gson by default. Before, you didn't need to worry about any integrated converter and you could use Gson out of the box. This library change affects your app and you need to import a converter as a sibling package as well. We'll touch the converter later within this chapter and show you how to config the Gson or any other response converter for your app.

**Converters**

```
compile 'com.squareup.retrofit2:converter-gson:2.0.0'
```

Also RxJava isn't integrated by default anymore. You need to add this additional import to your app's dependencies to get the reactive functionality back in the app.

**RxJava**

```
compile 'com.squareup.retrofit2:adapter-rxjava:2.0.0'
compile 'io.reactivex:rxandroid:1.1.0'
```

## RestAdapter —> Retrofit

The previously named `RestAdapter` class is renamed to `Retrofit`. The builder pattern is still available and you can easily chain available methods to customize the default behavior.

**Retrofit 1.9**

```
RestAdapter.Builder builder = new RestAdapter.Builder();
```

**Retrofit 2.x**

```
Retrofit.Builder builder = new Retrofit.Builder();
```

## setEndpoint —> baseUrl

You already read about the renaming of `RestAdapter` to `Retrofit`. There is another change within the `Retrofit` class which affects the base url (previously named endpoint url). Within Retrofit 1, the `setEndpoint(String url)` method defines the API's base url which is used later when defining partial routes within the interface declaration.

**Retrofit 1.9**

```
RestAdapter adapter = new RestAdapter.Builder()
    .setEndpoint(API_BASE_URL);
    .build();

YourService service = adapter.create(YourService.class);
```

Within Retrofit 2, the method is renamed to `baseUrl(String url)`. It still defines the base url for your API.

**Note:** Before you can call the `build()` method on the `Retrofit.Builder`, you need at least define the base url.

**Retrofit 2.x**

```
Retrofit retrofit = Retrofit.Builder()
    .baseUrl(API_BASE_URL);
    .build();

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

There is another major change in the API url handling. The next section explains the changes in more detail.

## Base Url Handling

There is a completely new url handling within Retrofit 2. This is very important to understand when updating from `1.x` to `2.x`!

Previously, the defined endpoint was always used as the default url for requests. Within your interface declaration which represent the individual API endpoints, you defined your partial routes including query or path parameters, request body or multiparts.

The API endpoint url and the partial url then were concatenated to the final url where the request is sent. To illustrate all theory, let's look at an example.

**Retrofit 1.x**

```
public interface UserService {
    @POST("me")
    User me();
}

RestAdapter adapter = RestAdapter.Builder()
    .baseUrl("https://your.api.url/v2/");
    .build();

UserService service = adapter.create(UserService.cass);

// the request url for service.me() is:
// https://your.api.url/v2/me
```

Within Retrofit `2.x`, you need to adjust your mind when it comes to API base urls. Retrofit `1` just concatenated the defined string values which resulted in a final request url. This behavior changes in Retrofit `2` since now the request url is created using the `HttpUrl.resolve()` method. This will create links similar to the well known `<a href>`.

To get things straight, look at the following code snippet which illustrates new way to resolve urls.

**Retrofit 2.x**

```
public interface UserService {
    @POST("/me")
    User me();
}

Retrofit retrofit = Retrofit.Builder()
    .baseUrl("https://your.api.url/v2");
    .build();

UserService service = retrofit.create(UserService.cass);

// the request url for service.me() is:
// https://your.api.url/me
```

You see, the leading `/` within the partial url overrides the `/v2` API endpoint definition. Removing the `/` from the partial url and adding it to the base url will bring the expected result.

```
public interface UserService {
    @POST("me")
    User me();
}

Retrofit retrofit = Retrofit.Builder()
    .baseUrl("https://your.api.url/v2/");
    .build();

UserService service = retrofit.create(UserService.cass);

// the request url for service.me() is:
// https://your.api.url/v2/me
```

**Pro Tip**: use relative urls for your partial endpoint urls and end your base url with the trailing slash `/`.

## Dynamic Urls

This is one of the features you won't immediately have a use-case in mind. However, we can provide you a use-case which we experienced during the implementation of a feature within one of our apps. We wanted to download a `.zip` file from an internet source and the files will have different urls. The files are either stored on Amazon's S3 or somewhere else on the web. Our problem was, that we need to create a `RestAdapter` with the respective base url every time we wanted to load a file. This gives you headache because you need to instantiate multiple HTTP clients and this is definitely bad practice.

Finally, the painful time will find its end and with Retrofit 2 we can use dynamic urls for an endpoint. You can leave the HTTP verb annotation empty and use the `@Url` annotation as a method parameter. Retrofit will map the provided url string and do the job for you.

```
public interface UserService {
    @GET
    public Call<File> getZipFile(@Url String url);
}
```

## OkHttp Required

You've seen at the beginning of this chapter, that you need to import OkHttp besides Retrofit itself. Retrofit 2 relies on OkHttp as the HTTP client and you need to import the library as well. Within Retrofit 1, you could set OkHttp manually as the HTTP client of choice. Now, OkHttp is required to use the `Call` class where responses get encapsulated.

```
compile 'com.squareup.okhttp:okhttp:3.2.0'
```

or

```
<dependency>
  <groupId>com.squareup.okhttp</groupId>
  <artifactId>okhttp</artifactId>
  <version>3.2.0</version>
</dependency>
```

## Interceptors Powered by OkHttp

Interceptors are a powerful way to customize requests with Retrofit. This feature was beneficial in Retrofit 1 and so will it be in version 2. A common use-case where you want to intercept the actual request is to add individual request headers. Depending on the API implementation, you'll want to pass the auth token as the value for the `Authorization` header.

Since Retrofit heavily relies on OkHttp, you need to customize the `OkHttpClient` and add an interceptor. The following code snippet illustrates how to add a new interceptor which uses the adds the `Authorization` and `Accept` headers to original request and proceeds with the actual execution.

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

        // Customize the request
        Request request = original.newBuilder()
                .header("Accept", "application/json")
                .header("Authorization", "auth-token")
                .method(original.method(), original.body())
                .build();

        Response response = chain.proceed(request);

        // Customize or return the response
        return response;
    }
});

Retrofit retrofit = Retrofit.Builder()
```

```
    .baseUrl("https://your.api.url/v2/");
    .client(client)
    .build();
```

If you're using a custom `OkHttpClient`, you need to set the client within the `Retrofit.Builder` by using the `.client()` method. This will update the default client with the enhanced self-made version.

You can apply the interceptors for several use-cases like authentication, logging, request and response manipulation, and many more.

## Synchronous & Asynchronous Requests

If you worked with Retrofit 1, you're familiar with the different method declaration in the service interface. Synchronous methods required a return type. In contrast, asynchronous methods required a generic `Callback` as the last method parameter.

Within Retrofit 2, there is no differentiation anymore for synchronous and asynchronous requests. Requests are now wrapped into a generic `Call` class using the desired response type. The following paragraphs will show you the differences of Retrofit 1 vs. Retrofit 2 in respect to service declaration and request execution.

### Interface Declaration

To differentiate a synchronous from an asynchronous request, you defined either the actual type or `void` as the response type. The latter required you to pass a generic `Callback` as the last method parameter. The following code snippet shows an exemplary interface declaration in Retrofit 1.

**Retrofit 1.9**

```
public interface UserService {
    // Synchronous Request
    @POST("/login")
    User login();

    // Asynchronous Request
    @POST("/login")
    void getUser(@Query String id, Callback<User> cb);
}
```

Within Retrofit 2, there is no different declaration anymore. The actual execution type is defined by the used method of the final `Call` object.

**Retrofit 2.x**

```java
public interface UserService {
    @POST("/login")
    Call<User> login();
}
```

## Request Execution

Retrofit 1 handles the request execution by using either a return type for synchronous or a `Callback` for asynchronous ones. If you worked with Retrofit before, you're familiar with the following code block.

**Retrofit 1.9**

```java
// synchronous
User user = userService.login();

// asynchronous
userService.login(new Callback<User>() {
@Override
    public void success(User user, Response response) {
        // handle response
    }

    @Override
    public void failure(RetrofitError error) {
        // handle error
    }
});
```

There is a completely different request execution handling in Retrofit 2. Since every request is wrapped into a `Call` object, you're now executing the request using either `call.execute()` for synchronous and `call.enqueue(new Callback<>() {})` for asynchronous ones. The example code below shows you how to perform each request type.

**Retrofit 2.x**

```
// synchronous
Call<User> call = userService.login();
User user = call.execute();

// asynchronous
Call<User> call = userService.login();
call.enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        User user = response.getBody();
    }

    @Override
    public void onFailure(Call<User> call, Throwable t) {
        // handle error
    }
}
```

<aside class="aside-info"> Synchronous requests on Android might cause app crashes on Android 4.0+. Use asynchronous request to avoid blocking the main UI thread which would cause app failures. </aside>

## Cancel Requests

With Retrofit 1, there was no way to cancel requests even if they weren't executed yet. This changes in Retrofit 2 and you're finally able to cancel any request if the HTTP scheduler didn't executed it already.

```
Call<User> call = userService.login();
User user = call.execute();

// changed your mind, cancel the request
call.cancel();
```

Doesn't matter if you're performing a synchronous or asynchronous request, OkHttp won't send the request if you're changing your mind fast enough.

## No Default Converter

The previous version 1 of Retrofit shipped with Gson integrated as the default JSON converter. The upcoming release won't have any default converter integrated anymore. You need to define your preferred converter as a dependency within your project. If you want to use Gson, you can use the following gradle import to define the sibling module:

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

There are multiple other converters available. The following list shows the required imports to get the updated converters for Retrofit 2.

**Available Converters**

- **GSON:** `com.squareup.retrofit2:converter-gson:2.0.0`
- **Moshi:** `com.squareup.retrofit2:converter-moshi:2.0.0`
- **Jackson:** `com.squareup.retrofit2:converter-jackson:2.0.0`
- **SimpleXML:** `com.squareup.retrofit2:converter-simplexml:2.0.0`
- **ProtoBuf:** `com.squareup.retrofit2:converter-protobuf:2.0.0`
- **Wire:** `com.squareup.retrofit2:converter-wire:2.0.0`

If none of the above listed converters fit your need, you can create your own one by implementing the abstract class `Converter.Factory`. In case you need help, lean on the available Retrofit converter implementations[55].

## Add Converter to Retrofit

We need to manually add the desired converters to Retrofit. The section above describes how to import a given converter module and additionally, you need to plug one ore many `ConverterFactory`'s into the `Retrofit` object.

```
Retrofit retrofit = Retrofit.Builder()
    .baseUrl("https://your.api.url/v2/");
    .addConverterFactory(ProtoConverterFactory.create())
    .addConverterFactory(GsonConverterFactory.create())
    .build();
```

The code example above plugs two converters to Retrofit. **The order in which you specify the converters matters!** Let's assume the following scenario which clarifies the importance of converter order. Since protocol buffers may be encoded in JSON, Retrofit would try to parse the data with Gson if it was defined as the first converter. But what we want is that the proto converter tries to parse the data first and if it can't handle it, pass it to the next converter (if there is another one available).

The names like `ConverterFactory` are not final and they'll probably change until the stable `2.0` is available. The current names sound more like Enterprise Java classes than they actually are. Of course we'll update this guide with future releases of Retrofit to keep it up-to-date with the current naming.

[55] https://github.com/square/retrofit/tree/master/retrofit-converters

## RxJava Integration

Retrofit 1 already integrated three request execution mechanisms: synchronous, asynchronous and RxJava. Within Retrofit 2, only synchronous and asynchronous requests are still available by default. Hence, the Retrofit development team created a way to plug additional execution mechanisms into Retrofit. You're able to add multiple mechanisms to your app like RxJava or Futures.

To get the RxJava functionality back into Retrofit 2, you need to import the following two dependencies. The first dependency is the RxJava `CallAdapter` which lets Retrofit know that there is a new way to handle requests. Precisely, that means you can exchange the `Call<T>` by `CustomizedCall<T>`. In case of RxJava, we'll change `Call<T>` with `Observable<T>`.

The second dependency is required to get the `AndroidSchedulers` class which is needed to subscribe code on Android's main thread.

**Gradle Dependencies**

```
compile 'com.squareup.retrofit2:adapter-rxjava:2.0.0'
compile 'io.reactivex:rxandroid:1.0.1'
```

The next thing required is to add the new `CallAdapter` to the `Retrofit` object before creating the service instance.

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

Code is better than all the theory. That's why we touch the next code example to illustrate the changes when using RxJava. At first, we declare a service interface. Afterwards, we assume to have a `userService` instance created and can directly leverage the Observable to observe on Android's main thread. We also pass a new Subscriber to the `subscribe` method which will finally provide the successful response or error.

```java
public interface UserService {
    @POST("/me")
    Observable<User> me();
}

// this code is part of your activity/fragment
Observable<User> observable = userService.me();
observable.observeOn(AndroidSchedulers.mainThread())
                          .subscribe(new Subscriber<User>() {
    @Override
    public void onCompleted() {
        // handle completed
    }

    @Override
    public void onError(Throwable e) {
        // handle error
    }

    @Override
    public void onNext(User user) {
        // handle response
    }
});
```

## No Logging

Actually, there is also sad news: no logging in Retrofit 2 anymore. The development team removed the logging feature. To be honest, the logging feature wasn't that reliable anyway. Jake Wharton explicitly stated that the logged messages or objects are the assumed values and they couldn't be proofed to be true. The actual request which arrives at the server may have a changed request body or something else.

Even though there is no integrated logging by default, you can leverage any Java logger and use it within a customized OkHttp interceptor. We've demonstrated this part in Chapter 7 — Logging.

## Future Update: WebSockets in Retrofit 2.1

WebSockets won't be available with the first stable release of Retrofit 2. Since the HTTP layer was removed from Retrofit 2 and all HTTP related operations are handed off to OkHttp, the WebSocket feature is completely developed within OkHttp. There is currently an experimental WebSocket implementation[56] within the OkHttp project. Early adopters can already test the feature by using OkHttp `3.2.0`.

Retrofit will benefit from the WebSocket feature once it's stable, because OkHttp is the solid basis of Retrofit.

## Conclusion

This was kind of an extensive overview of remarkable changes from Retrofit 1 to Retrofit 2 which require your attention and hands on code. The official change log[57] depicts all new features, improvements and fixes for each release. If you're interested in a specific thing, just search within the log.

Enjoy the upgrade to Retrofit 2 and working with the next release. We definitely think all changes are well thought out and the breaks are worth to lift Retrofit to a new level.

[56]https://github.com/square/okhttp/tree/master/okhttp-ws
[57]https://github.com/square/retrofit/blob/master/CHANGELOG.md