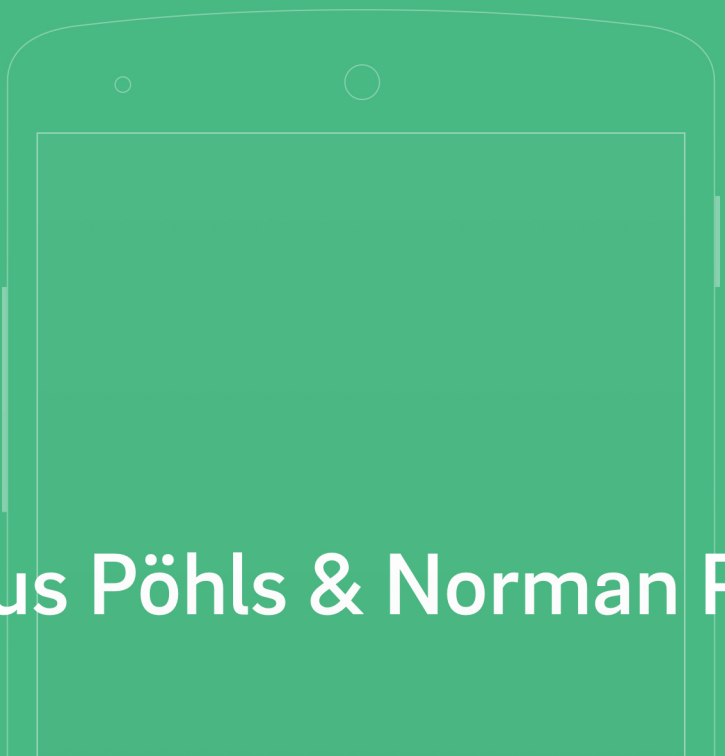
Retrofit

Love working with APIs on Android



Marcus Pöhls & Norman Peitek

Retrofit: Love Working with APIs on Android

You need to take delight building API clients on Android.

Marcus Pöhls

Also By Marcus Pöhls

Picasso: Easy Image Loading on Android

Glide: Customizable Image Loading on Android

Contents

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 and comments!

We keep the techy style from the tutorials to make this book a great resource for every developer working with Retrofit.

Who Is This Book For?

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

Rookie

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

Expert

You already worked with Retrofit before? You'll profit from our extensive code snippets and can improve your existing code base. Additionally, the book illustrates various use cases for different functionalities and setups like authentication against different backends, request composition and file uploads.

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.

The book starts out with an overview about what Retrofit is and how to prepare your Android project to use Retrofit. Furthermore, 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 the first client to perform a request against the GitHub API (learning by doing is helpful).

[1] https://futurestud.io/blog/retrofit-series-round-up/

After the jumpstart, we show you 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 or body as payload.

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 and how to use the refresh token to get a valid access token.

File handling can be kind of tricky, so let's take the road together! We guide you through the file handling with Retrofit and show the actions required to upload files.

Last but not least: release preparation for your app integrating Retrofit. 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.

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

Chapter 1 — Getting Started

Within this chapter we're going through the basics of Retrofit. Precisely, we start with a comprehensive overview of the Retrofit project. Afterwards, we show you how to prepare your Android project for the use of Retrofit, how JSON response mappings are performed and we create a sustainable Android API client.

What is Retrofit

The official description from the Retrofit website[2] depicts the library as follows:

> A type-safe REST client for Android and Java.

Retrofit transforms your REST API into a Java interface. You'll use annotations to describe HTTP requests: url parameter replacement and query parameter support are integrated by default, as well as functionality for multipart request body and file uploads.

Prepare Your Android 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 („Define Dependencies: Gradle or Maven"). If not, create a new Android project in your favorite IDE. We use Android Studio[3] and prefer Gradle as the build system, but you surely can use Maven as well.

Define Dependencies: Gradle or Maven

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

We propose using Retrofit with OkHttp, which needs to be defined as a dependency as well. OkHttp is an HTTP & SPDY client for Java and Android. It provides efficient network handling for your project, such as GZIP compression for file downloads, response caching for repeated requests, connection pooling for reduced latency, recovery in case of connection problems and more.

[2] http://square.github.io/retrofit/

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

At the time of this book being written, the latest Retrofit version is 1.9.0. There has been a lot activity in the Retrofit repository on GitHub during the last weeks. It looks like we're rapidly heading towards 2.0. We'll update this book to Retrofit 2.0 once it's released and has proven stabile. Of course, we'll notify and provide you with the free update of this book!

Ok, now let's add Retrofit (and OkHttp) as a dependency to our project:

pom.xml

```
<dependency>
    <groupId>com.squareup.retrofit</groupId>
    <artifactId>retrofit</artifactId>
    <version>1.9.0</version>
</dependency>
<dependency>
    <groupId>com.squareup.okhttp</groupId>
    <artifactId>okhttp</artifactId>
    <version>2.4.0</version>
</dependency>
```

build.gradle

```
dependencies {
    // rest client for http interaction and dependencies
    compile 'com.squareup.retrofit:retrofit:1.9.0'
    compile 'com.squareup.okhttp:okhttp:2.4.0'
}
```

Now that your project integrates Retrofit, let's have a look at how to define a Java interface mapping to your API for requests.

Internet Permission in AndroidManifest.xml

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. Add the following line within the manifest definition:

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

A common practise is, to add your app permissions as the first elements in your manifest file.

How to Declare an API Interface

Let's directly jump in and use a code example to make things approachable:

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

The snippet above defines a Java interface `TaskService` including one method `listTask`. The method and its parameters have Retrofit annotations which describe the behaviour of this method.

The `@GET()` annotation explicitly defines that a GET request will be performed on the method call. Further, the `@GET()` definition takes a string as the parameter representing the endpoint url of your API. Additionally, the endpoint url can be defined with placeholders which get substitued by path parameters.

The `@Path()` annotation for the `taskId` parameter maps the provided parameter value during the method call to the path within the request url. The declared `{id}` part within the endpoint url will be replaced by the provided value of `taskId`.

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/{myTaskId}/subtasks")` with method definition `List<Task> listSubTasks(@Path("id") String taskId);` won't map correctly since `myTaskId != id`.

The snippet above only describes how to define your API on the client side. Now we're interested in how to actually execute the API request. Therefore, we use Retrofit's `RestAdapter` class and create the client based on our API interface. First, we need to initialize our `RestAdapter` with the base url of our API endpoint. Afterwards, we use the previously created `RestAdapter` object to create the actual service client from our interface definition.

```
RestAdapter restAdapter = new RestAdapter.Builder()
        .setEndpoint("https://api.doyourtasks.com")
        .build();

TaskService service = restAdapter.create(TaskService.class);
```

The `restAdapter.create()` statement returns an implementation for our `TaskService` providing the defined methods. We can use the `listSubTasks()` method to request all subtasks for a given task.

```
List<Task> task = service.listSubTasks("your-task-id");
```

The statement above executes a synchronous request against your API endpoint located at `https://api.doyourtask task-id}/subtasks`. We'll cover the topic of synchronous and asynchrous requests with Retrofit later in this book. For now, be aware that Android in version 4.0 and later will crash when your app performs HTTP requests on the main thread.

Don't Use Synchronous Requests on Android's Main UI Thread

Synchronous requests can be the cause of app crashes on Android equal to or greater than 4.0. You're going to get a `android.os.NetworkOnMainThreadException` error and your app will stop working.

The basics are covered, we're ready to create a lasting Android API client!

JSON Response Mapping

Retrofit ships with Google's GSON by default. All you need to do is define the class of your response object and the response will be mapped automatically. Easy going, heh?

The example above used `List<Task>` as the return type of the `listSubTasks()` method. The `Task` class contains the properties' definition and GSON will map the JSON response data to an object of the class `Task`.

Sustainable Android REST Client

During the research for already existing Retrofit clients, the example repository of Bart Kiers[4] 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 stable foundation and extend it during future chapters within this book with further authentication functionality.

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

Service Generator

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

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

```
public class ServiceGenerator {
    private static final String BASE_URL = "https://your.base.url"
    private static RestAdapter.Builder builder =
        new RestAdapter.Builder()
            .setEndpoint(baseUrl)
            .setClient(new OkClient(new OkHttpClient()));

    public static <S> S createService(Class<S> serviceClass) {
        RestAdapter adapter = builder.build();
        return adapter.create(serviceClass);
    }
}
```

The `ServiceGenerator` class uses Retrofit's `RestAdapter`-Builder to create a new REST client with a given API base url. For example, GitHub's API base url is `https://developer.github.com/v3/` and you must update the provided example url with your own one.

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 examplary GitHub client.

Retrofit in Use

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

GitHub Client

The following code defines the `GitHubClient` and a method to request the list of contributors for a repository. It also illustrates the usage of Retrofit's parameter replacement functionality ({owner} and {repo} in the defined path will be replaced with the given variables when calling the object method).

```
public interface GitHubClient {
    @GET("/repos/{owner}/{repo}/contributors")
    List<Contributor> contributors(
            @Path("owner") String owner,
            @Path("repo") String repo
    );
}
```

There is a defined class `Contributor`. This class comprises the required object properties to map the response data.

```
static class Contributor {
    String login;
    int contributions;
}
```

With regard to the previous mentioned JSON mapping: the created `GitHubClient` defines a method named `contributors` with the return type `List<Contributor>`. Retrofit ensures the server response gets mapped correctly (in case the response matches the given class).

API Example Request

The snippet below illustrates the usage of `ServiceGenerator` to instantiate your client, specifically the `GitHubClient`, and the method call to get contributors via that client. This snippet is a modified version of provided Retrofit github-client example[5].

```
@Override
protected void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String API_URL = "https://developer.github.com/v3/";

    // Create a simple REST adapter which points to GitHub's API
    GitHubClient client =
            ServiceGenerator.createService(GitHubClient.class, API_URL);

    // Fetch and print a list of the contributors
    // for ‚android-boilerplate‘ from ‚fs-opensource‘
    List<Contributor> contributors =
            client.contributors("fs_opensource", "android-boilerplate");

    for (Contributor contributor : contributors) {
        System.out.println(
                contributor.login + " (" + contributor.contributions + ")");
    }
}
```

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

[5] 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 from authentication.

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

Additional Chapter Resources

- Retrofit Project Homepage[6]
- Retrofit API declaration documentation[7]
- Retrofit based authentication client[8] by Bart Kiers
- Retrofit Examples[9] by Square
- Android Studio IDE[10]

[6] http://square.github.io/retrofit/

[7] http://square.github.io/retrofit/#api-declaration

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

[9] https://github.com/square/retrofit/tree/master/samples

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

Chapter 2 — Requests

Retrofit offers a variety of capabilities to manipulate requests and their parameters. With these capabilities, you adjust the requests to your needs with annotations and placeholders and Retrofit handles everything for you. Within this chapter, we dive into the manipulation of your request url, path and query parameters. 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 and how to debug requests on the Android side.

We already know Retrofit's annotation style from the previous Getting Started chapter. However, let's quickly refresh how to declare an API on Android with annotations in the next paragraph.

API Delcaration With Retrofit

Retrofit uses annotations on Java interface methods to describe the handling of requests. One of the first things you want to define is the HTTP's 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. An example will clarify all theory:

```
1  public interface TaskService {
2      @GET("/relative/url/path") // <-- this is the important part
3      void method();
4  }
```

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 `RestAdapter` class integrates the base url and the defined resource url gets concatenated with the base url. Retrofit composes the final request url out of both string values.

Let's move on to query parameters as a way to add filter values to your requests.

Query Parameter and Multiple Parameters with the Same Name

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

Query Parameters

Query parameters are a common way to pass simple data from clients to servers. Common 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 hard coded query parameters within the resource endpoint url. Let's use the example below which **always** requests the task with `id=123` from our example API.

```
1  public interface TaskService {
2      @GET("/tasks?id=123")
3      void getTask123();
4  }
```

Our imaginery example API will return the data for a single task with `id=123`.

The example above is quite uncommon in real world APIs. Usually you need more dynamic behaviour 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.

```
1  public interface TaskService {
2      @GET("/tasks")
3      Task getTask(@Query("id") long taskId);
4  }
```

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

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

Of course, you can add more than one url parameter to your request. We'll cover that 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")
    Task getTask(
            @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>
```

The next subsection 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. Regarding the previous example of requesting tasks from an API, we can extend the query parameter to accept a list with multiple task ids.

The request url we want to create should look like:

```
http://api.example.com/tasks?id=123&id=124&id=125
```

The expected server response should be a list of tasks with the given `ids=[123, 124, 125]` from url query parameters.

The Retrofit method to perform requests with multiple query parameters of the same name is done by providing a list of `ids` as a parameter. Retrofit concatenates the given list to multiple query parameters of same name.

```
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 will look like the example above.

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

Besides single static and dynamic query parameters, we've covered the definition of multiple query parameters with the same name. Sometimes your request parameters don't have an intented purpose on every request. You can define optional query parameters as well.

Optional Query Parameters

You've certainly seen various APIs and ways to manipulate their responses. A very common use case is the sorting option for API endpoints. You typically can specify the order of data returned from a request in multiple ways: newest first, oldest first, order by name ascending or descending, and more.

Almost always the sorting option is not required to execute a successful request. The backend integrates a default sorting and you can just use the data as is. The next paragraph describes how to use optional query parameter.

Optional Query Parameters

Let's say the following example defines our API client on Android and we don't want to pass a sorting on every request. Only if the user specifies a sorting style, we use the parameter.

```
1  public interface TaskService {
2      @GET("/tasks")
3      Task getTasks(@Query("sort") String order);
4  }
```

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

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

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

The upcoming example depicts the service definition with two parameters for `sort` and `page`. We must use `Integer` for the `page` parameter if we want it to be optional.

```
1  public interface TaskService {
2      @GET("/tasks")
3      void List<Task> getTasks(
4              @Query("sort") String order,
5              @Query("page") Integer page);
6  }
```

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

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

Path Parameter

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

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

The string value of your path parameter within the resource url must match the defined value for the `@Path(…)` 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` annotation for the additional parameter.

Synchronous and Asynchronous Requests

Retrofit supports synchronous and asynchronous request execution. Users define the concrete execution by setting a return type (**synchronous**) or not (**asynchronous**) to service methods.

Synchronous Requests

Synchronous requests are declared by defining a return type. The example below yields a list of tasks as the response when executing the method `getTasks`.

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

Synchronous methods are executed on Android's main thread. That means the UI is blocked during request execution and no interaction is possible for this period.

Don't execute synchronous methods on Android's Main Thread

Synchronous requests can be the reason of app crashes on Android equal or greater than 4.0. You're going to get a `android.os.NetworkOnMainThreadException` error.

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 must 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.

Asynchronous Requests

In addition to synchronous calls, Retrofit supports asynchronous requests out of the box. Asynchronous requests don't have a return type. Instead, the defined method requires a typed callback as the last method parameter.

```
1  public interface TaskService {
2      @GET("/tasks")
3      void getTasks(Callback<List<Task>> cb);
4  }
```

Retrofit performs and handles the method execution in a separate thread. You don't have to worry about AsyncTaks or running the request in a separate thread. Retrofit has you covered for asynchronous request execution. The `Callback` class is generic and maps your defined return type. Our example returns a list of tasks and the Callback does the mapping internally.

Get Results from Asynchronous Requests

Using asynchronous requests forces you to implement a `Callback` with two callback methods: `success(…)` and `failure(…)`. When calling the asynchronous `getTasks(…)` method from a service class, you must implement the `Callback` and define what should be done once the request finishes. The following code snippet illustrates an exemplary implementation.

```
taskService.getTasks(new Callback<List<Task>>() {
    @Override
    public void success(List<Task> tasks, Response response) {
        // do stuff with returned tasks here
    }

    @Override
    public void failure(RetrofitError error) {
        // you should handle errors, too!
    }
});
```

Get Raw HTTP Response

If you need the raw HTTP response object, just define it as the method's return type. The `Response` class applies to both methods — synchronous and asynchronous — like any other class.

```
public interface TaskService {
    // synchronous
    @GET("/tasks")
    Response getTasks();

    // asynchronous
    @GET("/tasks")
    void getTasks(Callback<Response> cb);
}
```

The `Response` object comprises the data returned by the server. You can access the response headers and body, the status code, the status reason phrase and the request url.

Send Objects in Request Body

Retrofit offers the ability to send data to the API within the request body. Objects can be specified for use as the HTTP request body by using the `@Body` annotation before the method parameter.

```
public interface TaskService {
    @POST("/tasks")
    void createTask(@Body Task task, Callback<Task> cb);
}
```

The defined `RestAdapter`'s converter (GSON by default) will map the defined object to JSON and Retrofit passes the JSON as the request body to the defined server.

Example

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

```
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 class, the object fields and values will be converted to JSON before sending the request.

```
Task task = new Task(10, "my task title");
taskService.createTask(task, new Callback<Task>() {…});
```

Calling the service method `createTask` will convert the properties of `task` into JSON representation. The JSON of `task` will look like this:

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

Sending objects within the request body is done via annotations to the method parameters. Retrofit internally handles the conversion into JSON and mapping of the object into the request body.

Add Custom Request Header

Working with APIs doesn't narrow down to request composition with query or path parameters and sending objects 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 annotaion can be either key-value-pair as one string or as a JSON object. Let's use two concrete examples to illustrate the definition options:

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

The example above shows the key-value-definition for the static header. Further, you can pass a JSON object 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}")
    Task getTask(@Path("task_id") long taskId);
}
```

Additionally, you can define static headers via the `intercept` method of Retrofit's `RequestInterceptor`. Adding a custom `RequestInterceptor` to your `RestAdapter` requires you to override a single method `intercept`. The `intercept` method provides a parameter which can be used to manipulate your request. The following snippet shows examplarey code to fully understand how to integrate your custom `RequestInterceptor` into the `RestAdapter`.

```
RequestInterceptor requestInterceptor = new RequestInterceptor() {
    @Override
    public void intercept(RequestFacade request) {
        request.addHeader("User-Agent", "Your-App-Name");
        request.addHeader("Accept", "application/vnd.yourapi.v1.full+json");
    }
};

RestAdapter restAdapter = new RestAdapter.Builder()
        .setEndpoint("https://api.github.com")
        .setRequestInterceptor(requestInterceptor)
        .build();
```

As you can see, the example above sets the `User-Agent` and `Accept` header fields to the respective values. These values are passed with every request which is executed using this `RestAdapter` and the integrated `RequestInterceptor`.

We'll explicitly use the request interceptor within the upcoming authentication chapters.

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")
    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[11].

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.

That's it. Retrofit simplifies header manipulation and allows to simply change them for separated requests when necessary.

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

Using Retrofit's Log Level to Debug Requests

After diving deep into the art of using Retrofit for network requests, we'll look in this post at the debugging capabilities of Retrofit.

Request Logging

As you can see by this book, we at Future Studio are fans of Retrofit. Just a few days ago, we utilized another great feature of Retrofit: the request logging.

If you're having a similar problem and trouble understanding why a request from your Android app does not work the way it is intended, follow this guide for easy trouble shooting (or, at the very least, to be able to blame the API devs).

Activating Retrofit Logging

The ability to view the requests and responses can be a nice assistent. However, it's deactivated by default. Luckily, enabling the logging feature is fast and easy:

```
1 RestAdapter.Builder builder = new RestAdapter.Builder()
2         .setEndpoint(API_LOCATION)
3         .setLogLevel(RestAdapter.LogLevel.FULL) // this is the important line
4         .setClient(new OkClient(new OkHttpClient()));
```

This is all you have to do. Now start the app or the tests and force an execution of the requests in question.

Using the Logs

After the requests were made, take a look at the Android Logcat of your testing device. Retrofit posts a bunch of interesting things:

```
D/Retrofit: ---> HTTP GET https://api.github.com/users/fs-opensource
D/Retrofit: Accept: application/json
D/Retrofit: ---> END HTTP (no body)
D/Retrofit: <--- HTTP 200 https://api.github.com/users/fs-opensource (603ms)
D/Retrofit: Server: GitHub.com
D/Retrofit: Date: Sun, 08 Feb 2015 15:54:08 GMT
D/Retrofit: Content-Type: application/json; charset=utf-8
D/Retrofit: Transfer-Encoding: chunked
D/Retrofit: Status: 200 OK
D/Retrofit: X-RateLimit-Limit: 60
D/Retrofit: X-RateLimit-Remaining: 52
D/Retrofit: X-RateLimit-Reset: 1423413742
D/Retrofit: Cache-Control: public, max-age=60, s-maxage=60
D/Retrofit: Last-Modified: Sun, 30 Nov 2014 11:41:40 GMT
D/Retrofit: ETag: W/"254c978a4ab750aeba947905249aa808"
D/Retrofit: Vary: Accept
D/Retrofit: X-GitHub-Media-Type: github.v3
D/Retrofit: X-XSS-Protection: 1; mode=block
D/Retrofit: X-Frame-Options: deny
D/Retrofit: Content-Security-Policy: default-src 'none'
D/Retrofit: Access-Control-Allow-Credentials: true
D/Retrofit: Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-OA
D/Retrofit: Access-Control-Allow-Origin: *
D/Retrofit: X-GitHub-Request-Id: D5D3DF9E:72C8:19080DB3:54D786A0
D/Retrofit: Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
D/Retrofit: X-Content-Type-Options: nosniff
D/Retrofit: Vary: Accept-Encoding
D/Retrofit: X-Served-By: 173530fed4bbeb1e264b2ed22e8b5c20
D/Retrofit: OkHttp-Selected-Protocol: http/1.1
D/Retrofit: OkHttp-Sent-Millis: 1423410846583
D/Retrofit: OkHttp-Received-Millis: 1423410846714
D/Retrofit: {"login":"fs-opensource","id":7794026,"avatar_url":"https://avatars.githubusercontent.com/u/7794026?v=3","gravatar_id":"",
D/Retrofit: <--- END HTTP (1245-byte body)
```

Retrofit's Logs using Log Level FULL

As you can see, this includes the entire request and response body. While this can be useful and necessary, the information can be too much and clutter up your log.

Knowing the Levels

Not to worry, Retrofit has different logging levels to match the amount of required information without blowing up your logs too much. Let's take a look at the various levels:

NONE

Description: No logging. Nothing, niente, nada, nichts.

BASIC

Description: Log only the request method, Url, response status code and execution time.

Example:

```
D/Retrofit: ---> HTTP GET https://api.github.com/users/fs-opensource
D/Retrofit: <--- HTTP 200 https://api.github.com/users/fs-opensource (850ms)
```

Retrofit's Logs using BASIC

HEADERS

Description: Log the basic information along with request and response headers.

Example:

```
D/Retrofit: ---> HTTP GET https://api.github.com/users/fs-opensource
D/Retrofit: <--- HTTP 200 https://api.github.com/users/fs-opensource (2603ms)
D/Retrofit: ---> HTTP GET https://api.github.com/users/fs-opensource
D/Retrofit: Accept: application/json
D/Retrofit: ---> END HTTP (no body)
D/Retrofit: <--- HTTP 200 https://api.github.com/users/fs-opensource (2226ms)
D/Retrofit: Server: GitHub.com
D/Retrofit: Date: Sun, 08 Feb 2015 15:59:21 GMT
D/Retrofit: Content-Type: application/json; charset=utf-8
D/Retrofit: Transfer-Encoding: chunked
D/Retrofit: Status: 200 OK
D/Retrofit: X-RateLimit-Limit: 60
D/Retrofit: X-RateLimit-Remaining: 44
D/Retrofit: X-RateLimit-Reset: 1423413742
D/Retrofit: Cache-Control: public, max-age=60, s-maxage=60
D/Retrofit: Last-Modified: Sun, 30 Nov 2014 11:41:40 GMT
D/Retrofit: ETag: W/"254c978a4ab750aeba947905249aa808"
D/Retrofit: Vary: Accept
D/Retrofit: X-GitHub-Media-Type: github.v3
D/Retrofit: X-XSS-Protection: 1; mode=block
D/Retrofit: X-Frame-Options: deny
D/Retrofit: Content-Security-Policy: default-src 'none'
D/Retrofit: Access-Control-Allow-Credentials: true
D/Retrofit: Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-
D/Retrofit: Access-Control-Allow-Origin: *
D/Retrofit: X-GitHub-Request-Id: D5D3DF9E:202A:179CC3E8:54D787D8
D/Retrofit: Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
D/Retrofit: X-Content-Type-Options: nosniff
D/Retrofit: Vary: Accept-Encoding
D/Retrofit: X-Served-By: 173530fed4bbeb1e264b2ed22e8b5c20
D/Retrofit: OkHttp-Selected-Protocol: http/1.1
D/Retrofit: OkHttp-Sent-Millis: 1423411159442
D/Retrofit: OkHttp-Received-Millis: 1423411159635
D/Retrofit: <--- END HTTP (-1-byte body)
```

Retrofit's Logs using HEADERS

HEADERS_AND_ARGS

Description: Log the basic information along with request and response objects via `toString()`.

Example:

```
D/Retrofit: ---> HTTP GET https://api.github.com/users/fs-opensource
D/Retrofit: Accept: application/json
D/Retrofit: ---> END HTTP (no body)
D/Retrofit: <--- HTTP 200 https://api.github.com/users/fs-opensource (1254ms)
D/Retrofit: Server: GitHub.com
D/Retrofit: Date: Sun, 08 Feb 2015 16:01:39 GMT
D/Retrofit: Content-Type: application/json; charset=utf-8
D/Retrofit: Transfer-Encoding: chunked
D/Retrofit: Status: 200 OK
D/Retrofit: X-RateLimit-Limit: 60
D/Retrofit: X-RateLimit-Remaining: 43
D/Retrofit: X-RateLimit-Reset: 1423413742
D/Retrofit: Cache-Control: public, max-age=60, s-maxage=60
D/Retrofit: Last-Modified: Sun, 30 Nov 2014 11:41:40 GMT
D/Retrofit: ETag: W/"254c978a4ab750aeba947905249aa808"
D/Retrofit: Vary: Accept
D/Retrofit: X-GitHub-Media-Type: github.v3
D/Retrofit: X-XSS-Protection: 1; mode=block
D/Retrofit: X-Frame-Options: deny
D/Retrofit: Content-Security-Policy: default-src 'none'
D/Retrofit: Access-Control-Allow-Credentials: true
D/Retrofit: Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-
D/Retrofit: Access-Control-Allow-Origin: *
D/Retrofit: X-GitHub-Request-Id: D5D3DF9E:72C9:1BB76CBC:54D78863
D/Retrofit: Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
D/Retrofit: X-Content-Type-Options: nosniff
D/Retrofit: Vary: Accept-Encoding
D/Retrofit: X-Served-By: b0ef53392caa42315c6206737946d931
D/Retrofit: OkHttp-Selected-Protocol: http/1.1
D/Retrofit: OkHttp-Sent-Millis: 1423411297716
D/Retrofit: OkHttp-Received-Millis: 1423411297938
D/Retrofit: <--- END HTTP (-1-byte body)
D/Retrofit: <--- BODY:
```

Retrofit's Logs using HEADERS_AND_ARGS

FULL

Description: Log the headers, body, and metadata for both requests and responses.

Example:

see above at ***Using the Logs***

Utilize Logging

We hope this gave you a short introduction into a feature of Retrofit, which can be tremendously helpful. Keep this in mind if you want to debug a specific request or just log all the networking your app is doing.

Wrap-Up

This chapter gave you a detailed overview on how to customize requests to your needs. We covered query and path parameters, how to send objects in the request body, how to use synchronous and asynchronous requests, how to define HTTP headers generally or specifically for requests and finally the usage of Retrofit's log level for debugging purposes.

The next chapter shows you how to customize responses and create your own response converter. Besides that, we explain the integration of XML responses with Retrofit and how to map them to Java objects.

Additional Chapter Resources

- Heroku API Content-Range[12] definition
- Retrofit JavaDoc[13]
- Retrofit LogLevel JavaDoc[14]

[12] https://devcenter.heroku.com/articles/platform-api-reference#ranges
[13] http://square.github.io/retrofit/javadoc/index.html
[14] http://square.github.io/retrofit/javadoc/retrofit/RestAdapter.LogLevel.html

Chapter 3 — Response Handling

Typically your requests are followed up with an response and that's the reason why we do all the request composition and manipulation. We want to get data to show 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, or whatever, you name it.

This chapter shows you how to change Retrofit's default JSON converter and how to create your custom response converter. Besides that, you'll see how to integrate an XML response converter with Retrofit and how to define your Java objects to map the data. Additionally, we cover how to create a mock client to work offline or imitate API functionality.

Define a Custom Response Converter

Retrofit ships with Google's JSON[15] by default. Every JSON mapping is done with the help of GSON. Sometimes, your framework or attitude requires to change the integrated JSON converter. One of the well known converters is Jackson[16] and we use it to define our custom response converter and illustrate the replacement of GSON with Jackson.

Existing Retrofit Converters

Besides GSON, Retrofit can be configured with various converters and content formats. The retrofit-converters[17] directory lists existing response converters:

- JSON (using Jackson[18])
- XML (using Simple[19])
- Protocol Buffers (using protobuf[20] or Wire[21])

If you want to use an existing converter provided within the square repositories, use gradle to integrate the library. They all follow the same naming.

[15] https://code.google.com/p/google-gson/
[16] http://jackson.codehaus.org/
[17]
[18] https://github.com/FasterXML/jackson
[19] http://simple.sourceforge.net/
[20] https://code.google.com/p/protobuf/
[21] https://github.com/square/wire

```
dependencies {
    compile 'com.squareup.retrofit:converter-<converter-name>:1.9.0'
}
```

Concrete examples for Jackson and SimpleXML:

```
dependencies {
    // e.g. Jackson
    compile 'com.squareup.retrofit:converter-jackson:1.9.0'

    // e.g. XML
    compile 'com.squareup.retrofit:converter-simplexml:1.9.0'
}
```

You probably know how to deal with protocol buffers. :)

Create Your Own Converter

If you need or want to create your own Retrofit response converter, you can follow the steps below. We'll use the original Jackson library (not Square's `converter-jackson` library) to illustrate the concrete definition and handling.

There Are Existing Retrofit Converters[22]

You can use the provided Retrofit converters directly by integrating the respective gradle dependency. The list above depicts existing converters which are shipped as a separate gradle dependency and can be easily integrated with Retrofit.

Jackson Gradle Dependency

At first, define Jackson as a dependency for your project. When this book was written, Jackson 2.x was the lastest version to integrate. Jackson 2.x ships with a different group and artifact id than previous Jackson 1.x. Add the required Maven repository and compile dependency.

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

```
repositories {
    maven {
        url "http://repository.codehaus.org/org/codehaus"
    }
}

dependencies {
    compile 'com.fasterxml.jackson.core:jackson-databind:2.4.3'
}
```

Implement Your Custom JSON Converter

The important part of replacing the JSON converter of Retrofit is to implement the `Converter` interface and override its two methods: `fromBody` and `toBody`. Both methods handle the conversion from and to JSON. The following code snippet illustrates an examplary implementation for our `CustomJacksonConverter`

```
public class CustomJacksonConverter implements Converter {
    private ObjectMapper mapper = new ObjectMapper();

    public CustomJacksonConverter() {
    }

    @Override
    public Object fromBody(TypedInput body, Type type)
            throws ConversionException {

        JavaType javaType = mapper.getTypeFactory().constructType(type);

        try {
            return mapper.readValue(body.in(), javaType);
        } catch (IOException e) {
            throw new ConversionException(e);
        }
    }

    @Override
    public TypedOutput toBody(Object object) {
        try {
            String charset = "UTF-8";
            String json = mapper.writeValueAsString(object);
            return new JsonTypedOutput(json.getBytes(charset));
```

```
26          } catch (IOException e) {
27              throw new AssertionError(e);
28          }
29      }
30
31      private static class JsonTypedOutput implements TypedOutput {
32          private final byte[] jsonBytes;
33
34          JsonTypedOutput(byte[] jsonBytes) { this.jsonBytes = jsonBytes; }
35
36          @Override public String fileName() { return null; }
37          @Override public String mimeType() {
38              return "application/json; charset=UTF-8";
39          }
40          @Override public long length() { return jsonBytes.length; }
41          @Override public void writeTo(OutputStream out)
42                  throws IOException { out.write(jsonBytes); }
43      }
44  }
```

The `JsonTypedOutput` class is used to set the correct MIME-type. Since the output type is JSON, the MIME-type should be `application/json`.

Set Your Custom JSON Converter

The JSON converter of Retrofit is integrated in the `RestAdapter`. Changing the converter can be done by calling the `setConverter()` method on your `RestAdapter` object. Further, when creating a rest client, you can integrate the Jackson converter by setting the converter directly during `RestAdapter`'s build process.

```
1  // Create our Converter
2  CustomJacksonConverter jacksonConverteronverter =
3          new CustomJacksonConverter();
4
5  // Build the Retrofit REST adaptor with the custom
6  // converter and pointing to the API base url
7  RestAdapter restAdapter = new RestAdapter.Builder()
8          .setConverter(jacksonConverter)
9          .setEndpoint("https://api.example.com/")
10         .build();
```

Great! Now, all of your created `RestAdapter` objects will use `CustomJacksonConverter` for JSON handling.

How to Integrate XML Converter

We're going to replace the integrated JSON converter with an XML converter. Therefore, we set a custom response converter to Retrofit's `RestAdapter` which does all the request and response handling. Within the next subsections, we'll show you how to integrate an XML converter to map HTTP responses from XML to Java objects.

Define Gradle Dependency

First stop is the `build.gradle` file. The Retrofit project already provides an XML converter and therefore, you don't have to create your own (of course you could). Have a look at the previous section above on how to create or define your custom response converter[23]. You'll find information about existing converters for Retrofit and how to integrate them into your project. However, let's quickly recap the integration:

Here we go, define the dependency in your `build.gradle`.

```
dependencies {
    // Retrofit XML converter (Simple)
    compile 'com.squareup.retrofit:converter-simplexml:1.9.0'
}
```

Now wait for Gradle to finish the sync proecess. The XML converter library is integrated and now we need to adjust the `RestAdapter` to convert XML instead of JSON.

XMLConverter feat. RestAdapter

We have to define the `SimpleXMLConverter` as the converter for the `RestAdapter`. The following example illustrates the composition to setup an HTTP client with the given API endpoint url and most importantly the XML converter integration.

```
RestAdapter adapter = new RestAdapter.Builder()
        .setClient(new OkClient(new OkHttpClient()))
        .setEndpoint(yourBaseUrl)
        .setConverter(new SimpleXMLConverter())
        .build();
```

Within the `.setConverter()` method, we override the default JSON converter (GSON). From now on, Retrofit tries to parse each response data from XML to Java objects.

[23] http://futurestud.io/blog/retrofit-replace-the-integrated-json-converter/

Objects Love Annotations

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 the `Task` class below and add annoations to map the corresponding `tasks.xml` file.

tasks.xml This XML file does not 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. Definitely, you can use different naming and just put the XML element name to the `@Element` annotation.

How to Mock API-Responses

Developing your app against a deployed backend requires you to always have internet connection with your phone. Additionally, the backend functionality must implement all the features you're currently implementing on the client side. Let's assume some typical situations: you have an appointment somewhere outside your daily work place and want to fill the commuting time work. At least in Germany, the internet connection during train rides is totally bad. You'll be wasting more time waiting for responses than actually working on your app.

This is your chance to use a mock client and fake API responses locally. This approach has of course its pros and cons. Advantages: No internet connection required, since there won't be any requests sent to the API. Disadvantages: 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 Mock Client

You have multiple options to mock your API requests with Retrofit. One of the approaches is to implement Retrofit's `Client` interface. The `Client` interface declares only one method `execute(Request request)` providing the actual request as a parameter and returns a `Response` object.

The following code illustrate an examplary `MockClient`:

```
1  public class MockClient implements Client {
2      @Override
3      public Response execute(Request request) throws IOException {
4          Uri uri = Uri.parse(request.getUrl());
5
6          String responseString = "";
7
8          if (uri.toString().contains("/task?page")) {
9              responseString = createTasks(10);
10         } else if (uri.toString().contains("/task")) {
11             responseString = createTask();
12         }
13
14         return new Response(
15                 request.getUrl(), // request url
16                 200, // status code
17                 "no reason", // reason phrase
18                 new ArrayList<Header>(), // response headers
```

```
                // response body
                new TypedByteArray(
                        "application/json", responseString.getBytes())
        );
    }
}
```

The idea of this mock client is to fake the actual API response and create data to test your implementation. Since the `MockClient` is now the central point around the data access, it can get kind of bloated over time. We differentiate the request by its url and generate a JSON response string, which gets returned wrapped in a Retrofit `Response` object. Additionally, the response contains the status code, reason phrase for correct error handling and a list of response headers.

The example above uses simple `if` statements to test whether a request url matches a given scheme. If it does, a specific method gets called which generates the data and converts the object into JSON representation. The code below shows the `createTasks` method from the first `if` statement. It only illustrates the previous explanations.

```
private String createTasks(int count) {
    List<Task> tasks = new ArrayList<>();

    for (int i = 0; i < count; i++) {
        Task task = new Task();
        task.setId(1);
        task.setTitle("task title");
        task.setDescription("task description");
        task.setPosition(1);

        tasks.add(task);
    }


    Gson gson = new Gson();
    return gson.toJson(tasks);
}
```

The `createTasks` method loops 10 times and creates 10 `Task` objects. These objects are added to a list of tasks and finally converted to a JSON string with the help of GSON.

Enhanced Usage of Request Data

The mock client gets the actual request as the parameter and you can use all the data included. To enhance the `MockClient` and differenciate between request methods like `GET`, `POST`, `PUT` and `DELETE`, go ahead and check against the value within the request with `request.getMethod().equals("GET")`.

A common use case is the creation of a new object. Naturally, you would think of a `POST` request to the API, which returns the newly created object in return, including additional properties like id or creation date. You can of course fake that behavior, too. Just check the request method and url and afterwards, create a new object based on the provided data within `request.getBody()`.

Let's check the code snippet to map the request body provided as `TypedOutput` into your object representation (in this case `Task`).

```
1  OutputStream stream = new OutputStream() {
2      public CharArrayWriter string = new CharArrayWriter();
3      @Override
4      public void write(int i) throws IOException {
5          this.string.append((char) i);
6      }
7
8      @Override
9      public String toString() {
10         return this.string.toString();
11     }
12 };
13 TypedOutput body = request.getBody();
14 body.writeTo(stream);
15 Gson gson = new Gson();
16 Task task = gson.fromJson(stream.toString(), Task.class);
```

First, we declare the `OutputStream` which translates the incoming stream of characters into a complete string representation. Afterwards, we write the request body into the `OutputStream` and finally, GSON handles the conversion from the received string into the object. Now you can use and adjust the objects as you choose.

Wrap-Up

Within this chapter, you've learned how to replace the integrated JSON response converter (GSON) with one of the available converters from Square's repository. Additionally, you got a detailed overview with code examples on how to create a custom response converter on your own. Retrofit also works fine with XML and — of course — works great with JSON. We integrated SimpleXML with Retrofit to parse XML responses correctly into Java objects.

Sometimes you need to fake API functionality locally to make progress with your app. The section on how to create a mock client to imitate API functionality provides an in-depth look on how to simulate the normal API request behavior within your app and further how to use requests and return mock responses.

Within the next chapter, we finally 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.

Additional Chapter Resources

- Simple XML Converter[24]
- Retrofit Response Converters[25]

[24] http://simple.sourceforge.net/download/stream/doc/tutorial/tutorial.php
[25] https://github.com/square/retrofit/tree/master/retrofit-converters

Chapter 4 — Authentication

We walked you through the basics of Retrofit and you know all important parts of how to create requests and respective handle reponses. Now we're going to face the most common example which is integrated in almost every Android app: authentication. We show you how to integrate three types of authentication within your app. Starting with basic authentication with a username and password, we're moving to token authentication and finally to OAuth handling on Android.

We've added an Android project within the extras of this book. You'll find the complete code within this project. Feel free to use the code within your Android app as well.

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 practise to let users authenticate initially with their credentials and the backend sends an access token in return for successful authentication. We'll dive into the usage of access tokens with Retrofit within the next section. For now, let's see how to integrate basic authentication with Retrofit.

Within the getting started post of this series, we created an initial version of the Android client to perform API/HTTP requests. We'll use the client foundation from the Getting Started chapter and enhance it with functionality for basic authentication. The code below shows the snippet to create the rudimentary client for any service:

```
1  String BASE_URL = "https://your.api.url";
2
3  RestAdapter.Builder builder = new RestAdapter.Builder()
4          .setEndpoint(BASE_URL)
5          .setClient(new OkClient(new OkHttpClient()));
6
7  RestAdapter adapter = builder.build();
```

You already know that Retrofit uses the `RestAdapter` and its integrated `Builder` to create new API clients on Android. The previous snippet doesn't has support to send user credentials with the request. The next subsection explains how to do it.

Integrate Basic Authentication

Since we're adding support for basic authentication, we need at least two new parameters: **username** and **password**. You can use the username parameter for email, too. The basic approach of creating the client is the same as in the rudimentary approach: use the `RestAdapter` class as a helper to set the endpoint url and create the OkHTTP client for any HTTP requests and response handling.

The difference now: we use a RequestInterceptor to set header values for the HTTP request to the given endpoint URL. However, this is only done if username and password are provided. If you don't provide username and password, the same client will be created as the first method did.

Here's the code snippet which enhances the `RestAdapter.Builder` to integrate with basic authentication.

```
1  public class ServiceGenerator {
2      public static final String BASE_URL = "https://your.api.url";
3
4      public static <S> S createService(
5              Class<S> serviceClass, String username, String password) {
6
7          // set endpoint url and use OkHTTP as HTTP client
8          RestAdapter.Builder builder = new RestAdapter.Builder()
9                  .setEndpoint(BASE_URL)
10                 .setClient(new OkClient(new OkHttpClient()));
11
12         if (username != null && password != null) {
13             // concatenate username and password with colon inbetween
14             String credentials = username + ":" + password;
15             // create Base64 encodet string
16             String string = "Basic " + Base64.encodeToString(
17                     credentials.getBytes(), Base64.NO_WRAP);
18
19             builder.setRequestInterceptor(new RequestInterceptor() {
20                 @Override
21                 public void intercept(RequestFacade request) {
22                     request.addHeader("Accept", "application/json");
23                     request.addHeader("Authorization", string);
24                 }
25             });
26         }
27
28         RestAdapter adapter = builder.build();
29         return adapter.create(serviceClass);
```

```
30      }
31  }
```

For the authentication part, we have to adjust the format of given username/email 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.

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 web service you're going to call with this client specifies another header field to expect the user's credentials, adjust the header field from **Authorization** to **your header field**.

The **Accept** header is important if you want to receive the server response in a specific format. In our example, we want to receive the response JSON formatted, since Retrofit ships with Google's GSON[26] to serialize objects into their JSON representation and vice versa.

Usage

For example, let's assume you defined a `LoginService` like the code below.

```
1  public interface LoginService {
2      @POST("/login")
3      User basicLogin();
4  }
```

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

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

```
1  LoginService loginService =
2          ServiceGenerator.createService(LoginService.class, username, password);
3  User user = loginService.login();
```

The `ServiceGenerator` method will create the HTTP client, including the set HTTP header field for authentication with your provided username and password. Once you call the `basicLogin` method of `loginService`, the provided credentials will be automatically passed to your defined API endpoint.

[26] https://code.google.com/p/google-gson/

Token Authentication

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

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 handling token authentication. Let's extend the `ServiceGenerator` with the following method:

```
1  public class ServiceGenerator {
2      public static final String BASE_URL = "https://your.api.url";
3
4      // returns an API client of type serviceClass
5      public static <S> S createService(
6              Class<S> serviceClass, final String token) {
7
8          RestAdapter.Builder builder = new RestAdapter.Builder()
9                  .setEndpoint(BASE_URL)
10                 .setClient(new OkClient(new OkHttpClient()));
11
12         if (token != null) {
13             builder.setRequestInterceptor(new RequestInterceptor() {
14                 @Override
15                 public void intercept(RequestFacade request) {
16                     request.addHeader("Accept", "application/json");
17                     request.addHeader("Authorization", token);
18                 }
19             });
20         }
21
22         RestAdapter adapter = builder.build();
23         return adapter.create(serviceClass);
24     }
25 }
```

As you can see, we pass the token value as a `String` variable into the method and use the `RequestInterceptor` to set the HTTP header field for **Authorization**. If you're using another HTTP

header field for your authentication token, either adjust the code above or create a new method and pass the header field name as a parameter to this method.

That's it! :)

Now, every HTTP client created with this method integrates the token value for the **Authorization** header field and automatically passes the token value to your API endpoint with any request.

Example Usage

Let's create an example and see some code. The `UserService` interface below declares a method called `me()`. This example method returns a user object created from the API response.

```
1  public interface UserService {
2      @POST("/me")
3      User me();
4  }
```

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.

```
1  UserService userService =
2          ServiceGenerator.create(UserService.class, token);
3  User user = userService.me();
```

This code illustrates how to use the presented classes. You have to replace some variables with your correct values.

OAuth on Android With Retrofit

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 acces to your data.

This post won't go into much detail about OAuth itself. It simply presents the basic principles and necessary details to understand the authentication flow.

OAuth Basics

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

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 to interact with the API.

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 a **client secret**. Both values are required to authenticate your app against the service or API.

Create Your Project

We'll assume you already have an existing Android project. If you don't, just go ahead and create an Android project from scratch. When you're done, go ahead 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. The snippet below shows the required method within the `ServiceGenerator` class. That doesn't mean you should delete the previously created method(s) for basic authentication, since you'll need them for OAuth as well.

```
public class ServiceGenerator {
    private static RestAdapter.Builder builder =
            new RestAdapter.Builder().setClient(new OkClient(new OkHttpClient()))\
);

    public static final String BASE_URL = "https://your.api.url";

    // method for basic auth
    // we saved space and collapsed the method as in your IDE
    public static <S> S createService(
            Class<S> serviceClass, String username, String password) {…}

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

        // set endpoint url
        builder.setEndpoint(baseUrl);

        if (accessToken != null) {
            builder.setRequestInterceptor(new RequestInterceptor() {
                @Override
                public void intercept(RequestFacade request) {
                    request.addHeader("Accept", "application/json");
                    request.addHeader("Authorization",
                            accessToken.getTokenType() + " " +
                                    accessToken.getAccessToken());
                }
            });
        }

        RestAdapter adapter = builder.build();
        return adapter.create(serviceClass);
    }
}
```

We're using the RequestInterceptor to set the **Authorization** field within the HTTP request header. This field consists of two parts: first, the token type which is **Bearer** for OAuth requests and second, the access token.

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

```
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 tokenType;
    }
}
```

The `AccessToken` class consists of two fields: `accesToken` and `tokenType`. Since OAuth API implementations require the token type to be in uppercase, we check the styling first. If it doesn't fit, we update the style. 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. 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.

```
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
```

```
10      protected void onCreate(Bundle savedInstanceState) {
11          super.onCreate(savedInstanceState);
12          setContentView(R.layout.activity_login);
13
14          Button loginButton (Button) findViewById(R.id.loginbutton);
15          loginButton.setOnClickListener(new View.OnClickListener() {
16              @Override
17              public void onClick(View v) {
18                  Intent intent = new Intent(
19                          Intent.ACTION_VIEW,
20                          Uri.parse(
21                                  ServiceGenerator.BASE_URL + "/login" +
22                                          "?client_id=" + clientId +
23                                          "&redirect_uri=" + redirectUri));
24
25                  startActivity(intent);
26              }
27          });
28      }
29  }
```

You must adjust the values for class properties **clientId**, **clientSecret**, **redirectUri**. Further, we set an OnClickListener for the defined login button within the `onCreate` method. Once the onClick event gets triggered, it creates a new intent showing a webview for the defined Uri. **Important**: you have to provide your client id and client secret in this request, since the API requires the two parameters for further processing including the app you're using.

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

The layout for `activity_login.xml` can look like this.

```
1   <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2       xmlns:tools="http://schemas.android.com/tools"
3       android:layout_width="match_parent"
4       android:layout_height="match_parent"
5       >
6       <Button
7           android:layout_width="match_parent"
8           android:layout_height="wrap_content"
9           android:text="Login"
10          android:id="@+id/loginbutton"
11          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 intent's action, category and data.

The intent filter is required to make Android return to your app, so you can grab further 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 acitvity.

```
<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 defitions inside your `AndroidManifest.xml` file. Let's just glance over the snippet above and clarify the intent filter. 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 develop the app for. The redirect url gets called once the login procedure finishes successfully and includes the authorization code. This code is required to finally request the access token.

Catch the Authorization Code

Ok, until here we have defined the intent to show the web view which presents as a **deny** or **allow** view. You'll notice the style of this view when seeing it.

Now we want to get the access token for further API interaction. 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 since it's easier to explain the contents and you don't get overloaded with duplicated code.

```
@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("")) {
            // 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 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`.

```
public interface LoginService {
    @POST("/login")
    User basicLogin();

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

This is the interface definition which is passed to `ServiceGenerator` to create a Retrofit HTTP client. The `getAccessToken` method expects two query parameters: the `authCode` and a `grantType`. The authorizaton code is the one we got from our first request and the grant type is probably `authorization_code` since the API runs on a web server. There are 4 grant types: **authorization code** for apps running on web servers, **implicit** for mobile or browser based apps, **password** for authentication with username and password, **client credentials** for app access.

However, have a look at the API documentation of the service you're developing the app for.

Now the complete code for `onResume` to get the token.

```
@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");
```

```
13              if (code != null) {
14                  // get access token
15                  LoginService loginService =
16                          ServiceGenerator.createService(
17                                  LoginService.class, clientId, clientSecret);
18
19                  AccessToken accessToken =
20                          loginService.getAccessToken(code, "authorization_code");
21              } else if (uri.getQueryParameter("error") != null) {
22                  // handle error here
23              }
24          }
25      }
26  }
```

And you're done. You probably have to adjust the **grant type** value for the API you're reqesting. In case everything went fine, you'll get the access token in return and can perform further API requests using this token.

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

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

OAuth Refresh-Token Handling

When working with OAuth APIs, you sometimes have to manage access tokens and also refresh tokens. Generally, the access token is what you'll use to authenticate your requests. The tokens are short lived and the lifetime differs per API provider. Validity ranges are between one hour and infinite (if there is no refresh token handling implemented).

In contrast, the request token has a much longer lifetime. Sometimes it's infinite and you have to revoke the access token manually.

The idea to have two tokens for OAuth is in case your access token got compromised, the attacker has a limited time window to probably misuse it. The refresh token is useless, since the attacker needs the client id and client secret to request a new access token.

Recognize Invalid Access Token

Besides security concerns, from developer perspective it's important 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).

Refresh Invalid Access Token

However, the response will probably include an error description. You should check the developer documentation of your API provider how they handle expired access tokens and what comes in return. If your error descriptions contain a sentence like "Access token expired" or just action words like "AccessTokenExpiration" you can use string comparison or checks if a keyword is contained to kickoff the correct handling.

Actually, the way to go is having a procedure in place to refresh the invalid access token in background and only show errors to the user if something goes wrong badly.

Let's assume your app's start activity shows a timeline of posts where authentication is required to view the posts. The timeline data gets requested using the access token. You didn't start the app a while and the latest access token is expired. Starting your app and requesting the timeline data will result in an error and you have to interpret and handle this error. We assume the following snippet to be from your code and illustrate the error handling for the out of date access token. Furthermore, we also assume that the API returns the status code `401` for unauthorized access including expired access tokens.

```
1  private void getTimeline() {
2      timelineService.getTimeline(new Callback<Timeline>() {
3          @Override
4          public void success(Timeline timeline, Response response) {…}
5
6          @Override
7          public void failure(RetrofitError error) {
8              int status = error.getResponse().getStatus();
9              if (status == 401) {
10                 refreshAccessToken();
11             }
12         }
13     });
14 }
```

The snippet above describes the request to get the timeline data. We concentrate on the error handling since we simulate the failing request with the outdated access token. Within the `failure(RetrofitError error)` method, we first get the response status code. Afterwards, we check if our request failed due to unauthorized access and if so, we call the method `refreshAccessToken()`. This method should be inside your activity and it handles the refresh operation of the access token.

Once the refresh finished and you received a valid access token, jump back to the method which executes the request to load the timeline.

```
private void refreshAccessToken() {
    userService.refreshToken(…, new Callback<AccessToken>() {
        @Override
        public void success(AccessToken token, Response response) {
            // save the new access token
            // jump back to load the timeline
            getTimeline();
        }

        @Override
        public void failure(RetrofitError error) {
            // when the shit hits the fan
            // looks like something went completely wrong
        }
    });
}
```

The code example above assumes you defined the `refreshToken()` method within the `UserService` interface. Usually, the refresh operation requires additional parameters. Check the developer docs for further information about the data you need to send while refreshing the access token.

Let's face a common scenario within our Android apps and how to combine it with the refresh token method.

What do I do if there are multiple API calls within my activity and I want to jump back to the correct method in case a refresh failed and I needed to refresh the access token?

We all know this situation within our apps and it's — of course — very common to have multiple different requests within an activity. Actually, the situation is kind of tricky to handle. You have to save the lastly called request method to remember the entry point once the token refresh operation finished. You want to provide a seamless experience and the user shouldn't notice that his access token was expired and you needed to refresh it behind the scenes.

The extras to this book provide extensive code examples and we prepared one to illustrate the refresh token scenario. In case all the theory is boring and hard to follow, jump right into the code and check the authentication examples.

Wrap Up

The authentication sections comprise all the knowledge we gained through the previous chapters about how to assemble requests and consume responses. Further, it's a very common use case within Android apps and we hope you profit from all the code examples. We know this chapter is code-heavy and we intentionally provided all the snippets. We geeks need some code to understand the context.

Within the next chapter, we show you how to upload files with Retrofit and what configurations to keep in mind when working with files.

Chapter 5 — File Upload with Retrofit

Nowadays, file handling on the client side is an essential capability to enable features like file upload and 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.

How to Upload Files

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 guide can be used for any kind of file and not just images.

Upload Files with Retrofit

Retrofit provides the ability to perform multipart requests. This enables the feature to upload files to a server. To perform these kind of requests, the `@Multipart` annotation is required. Please have a look at the following code snippet, which shows the interface and method definition to upload a file.

```
1  public interface FileUploadService {
2      @Multipart
3      @POST("/upload")
4      void upload(@Part("myfile") TypedFile file,
5                  @Part("description") String description,
6                  Callback<String> cb);
7  }
```

This `FileUploadService` interface is the client side API declaration. You're going to request the partial endpoint url `/upload` hooked with two parameters: a file named `myfile` and a `description` of type String.

As you remember, to perform the actual request, you need to instantiate a `RestAdapter`. Have a look at our Retrofit Getting Started Guide[27] where we present a `ServiceGenerator` class to create services for defined interfaces.

[27] http://futurestud.io/blog/retrofit-getting-started-and-android-client/

The `ServiceGenerator` (described in the getting started post) creates and instantiates the required HTTP client for your service interface. In our case, we pass the `FileUploadService` to the static method `createService` and get a functional client implementation to perform the `upload` operation.

The following code snippet illustrates the usage of previously described methods:

```
1  FileUploadService service =
2          ServiceGenerator.createService(FileUpload.class);
3  TypedFile typedFile =
4          new TypedFile(
5                  "multipart/form-data",
6                  new File("path/to/your/file"));
7
8  String description = "hello, this is description speaking";
9
10 service.upload(typedFile, description, new Callback<String>() {
11     @Override
12     public void success(String s, Response response) {
13         Log.e("Upload", "success");
14     }
15
16     @Override
17     public void failure(RetrofitError error) {
18         Log.e("Upload", "error");
19     }
20 });
```

`TypedFile`[28] is a class from Retrofit representing a file with mime type. We use `multipart/form-data` as the mime type and create a file from a given path.

That's all the magic within your Android app. We initially had some issues performing the request, since we didn't pay enough attention to the content-type within our `ServiceGenerator`. When sending multipart data, **don't define `application/json` as content-type**.

Remember the Content-Type

Keep an eye on the content-type of the `RestAdapter`. If you specify the header `request.addHeader("Content-Type", "application/json");` via `RequestInterceptor`, the request will probably **fail** on the server side, since you're sending multipart data and not JSON.

[28] http://square.github.io/retrofit/javadoc/retrofit/mime/TypedFile.html

Example hapi.js Server to Handle File Uploads

We've tested the upload functionality from Android to the backend with a hapi.js server. For that, we implemented a rudimentary feature set on the server side to just log the incoming data. Multiparty[29] was our module of choice to parse incoming multipart data from clients. Of course, you can use whatever you prefer.

We just showed you the code of a servers's route configuration to get an impression how the file handling works on the server side. The complete example code is attached within the extras of this book.

The hapi.js route configuration for our file upload test:

```
server.route([{
    method: 'POST',
    path: '/upload',
    config: {
        payload: {
            maxBytes: 209715200,
            output: 'stream',
            parse: false
        },
        handler: function(request, reply) {
            var multipary = require('multiparty');
            var form = new multiparty.Form();
            form.parse(requset.payload, function(err, fields, files) {
                console.log(err);
                console.log(fields);
                console.log(files);
            });
        }
    }
}]);
```

We want to separate the topics of file upload with Retrofit on Android and the hapi.js server. That's why we just leave the code snippet above uncommented. Nevertheless, here is the server log for a successful request.

Server Log for incoming data

[29] https://www.npmjs.com/package/multiparty

```
null
{ description: [ 'hello, this is description speaking' ] }
{ myfile:
   [ { fieldName: 'myfile',
       originalFilename: 'IMG-20150311-WA0000.jpg',
       path: '/var/folders/rq/q_m4_21j3lqf1lw48fqttx_80000gn/T/wcvFPnGewEPCm9IHM\
7ynAS3I.jpg',
       headers: [Object],
       size: 34287 } ] }
```

The first `null` log is the `err` object. Afterwards, the `description` field and finally `myfile`.

Wrap-Up

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. 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.

The next chapter describes the configuration of ProGuard when using Retrofit within your app. Further, we give you a complete ProGuard configuration to copy and 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.

Chapter 6 — App Release Preparation

Google highly recommends to use ProGuard for release builds which 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 get a wrap up about ProGuard and afterwards we dive into examplary ProGuard rules to keep your app working after compiling it with ProGuard enabled.

Enable ProGuard

First things first: What is ProGuard[30]? 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. ProGuards only runs when you build your app in release mode and additionally you need to set the option `minifyEnabled` to `true`. The following code is an extract from the app's `build.gradle`.

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'progu\
ard-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.

[30] https://developer.android.com/tools/help/proguard.html

Configure ProGuard Rules for Retrofit

ProGuard is very harsh in regards to deleting code. Within many situations, ProGuard doesn't analyze 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 public class <MyClass>
```

We use the following snippet within our apps distributed via Google Play. Since Retrofit integrates GSON, you need to keep track of those classes as well. Copy the snippet to your `proguard-rules.pro` file to keep necessary Retrofit classes within your app.

```
# Retrofit

-keep class com.google.gson.** { *; }
-keep class com.google.inject.** { *; }

-keep class org.apache.http.** { *; }
-keep class org.apache.james.mime4j.** { *; }

-keep class javax.inject.** { *; }
-keep class javax.xml.stream.** { *; }

-keep class retrofit.** { *; }

-keep class com.google.appengine.** { *; }

-keepattributes *Annotation*
-keepattributes Signature

-dontwarn com.squareup.okhttp.*
-dontwarn rx.**

-dontwarn javax.xml.stream.**
-dontwarn com.google.appengine.**
-dontwarn java.nio.file.**
-dontwarn org.codehaus.**

```

```
27 # Add any classes the interact with gson
28 # the following line is for illustration purposes
29 -keep class io.futurestud.retrofitbook.android.services.models.User
```

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 comprehensable one. The respective file is located within `<sdk_root>/tools/proguard/` directory. The syntax to execute the `retrace` tool is this:

```
1 retrace.bat|retrace.sh [-verbose] mapping.txt [<stacktrace_file>]
```

Precisely:

```
1 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.

Wrap-Up

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.

Chapter 7 — Retrofit 2 Upgrade Guide from 1.x

Retrofit is a great HTTP client for Android (and Java) and the second major release will be available by the end of 2015. The previous chapters are all based on Retrofit 1.9 and due to the popularity of this book, we immediately decided to write this upgrade guide from `1.9` (or any other Retrofit `1.x` version) to `2.0`. 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 is getting close to the upcoming second major release in the end of this year. 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.

At the time of writing this article, the latest Retrofit 2 release is `2.0.0-beta2`. This release is officially declared as a pre-release, but you can use it within your production apps. However, even if you want to stay with Retrofit 1 as along as it is the stable and more robust release, you can branch out and start updating to the next Retrofit version while benefiting from all the improvements.

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

```
1  compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
2  compile 'com.squareup.okhttp:okhttp:2.5.0'
```

Maven: Retrofit & OkHttp

```
<dependency>
  <groupId>com.squareup.retrofit</groupId>
  <artifactId>retrofit</artifactId>
  <version>2.0.0-beta2</version>
</dependency>
<dependency>
  <groupId>com.squareup.okhttp</groupId>
  <artifactId>okhttp</artifactId>
  <version>2.5.0</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 post and show you how to config the Gson or any other response converter for your app.

Converters

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

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

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 behaviour.

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 post, 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:2.5.0'
```

or

```
<dependency>
  <groupId>com.squareup.okhttp</groupId>
  <artifactId>okhttp</artifactId>
  <version>2.5.0</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()
```

```
22      .baseUrl("https://your.api.url/v2/");
23      .client(client)
24      .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

```
1  public interface UserService {
2      // Synchronous Request
3      @POST("/login")
4      User login();
5
6      // Asynchronous Request
7      @POST("/login")
8      void getUser(@Query String id, Callback<User> cb);
9  }
```

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(Response<User> response, Retrofit retrofit) {
        User user = response.getBody();
    }

    @Override
    public void onFailure(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-beta2'
```

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.retrofit:converter-gson:2.0.0-beta2`
- **Moshi:** `com.squareup.retrofit:converter-moshi:2.0.0-beta2`
- **Jackson:** `com.squareup.retrofit:converter-jackson:2.0.0-beta2`
- **SimpleXML:** `com.squareup.retrofit:converter-simplexml:2.0.0-beta2`
- **ProtoBuf:** `com.squareup.retrofit:converter-protobuf:2.0.0-beta2`
- **Wire:** `com.squareup.retrofit:converter-wire:2.0.0-beta2`

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[31].

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.

[31] 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.retrofit:adapter-rxjava:2.0.0-beta2'
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'll definitely write another book chapter describing how to integrate a logger with Retrofit 2.

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[32] within the OkHttp project. Early adopters can already test the feature by using OkHttp `2.5`.

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[33] depicts all new features, improvements and fixes for each release. If you're interested in a specific thing, just search within the log.

Retrofit `2` is still in beta and you should stay with `1.9` for your production apps. If you want to prepare your project for the next release, you're definitely not wasting your time by branching out and adjusting your code. There might be minor method signature changes within the upcoming pre-releases of Retrofit, nonetheless the most methods will stay the same and you're on save routes already preparing for the update.

If you're more like a cutting edge guy, you're probably safe using it for your apps already. If you find any bug using the library, please file an issue[34] and help to push Retrofit to the next level. Also, if you have any concerns respective the internal API, raise your hand now! Once the second major release is released as stable, there won't be any breaking changes for a while.

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.

[32] https://github.com/square/okhttp/tree/master/okhttp-ws

[33] https://github.com/square/retrofit/blob/master/CHANGELOG.md

[34] https://github.com/square/retrofit/issues

Chapter 8 — Use OkHttp 3 with Retrofit 1.9

The latest release of Retrofit 1 (`1.9.0` at the time of writing this book) directly depends on OkHttp `2.2.0`. For Retrofit 2, the developers introduced multiple breaking changes, because Retrofit in its second major release will actually **use** OkHttp instead of working with and around it. That means, OkHttp got a lot attention and improvements in regard to functionality and stability. Also, the third major version was just released!

This chapter will guide you through the migration and setup of Retrofit `1.9` with OkHttp 3.

Dependencies

At first you need to add OkHttp 3 to your project. At the time of writing this chapter, OkHttp 3 was just released, that means we can directly use the fresh baked cookie :)

Furthermore, you'll need the wrapper class from Jake Wharton's `retrofit1-okhttp3-client`[35] repository on GitHub. Jake already released the client code and you can directly make use of this little helper by importing the Gradle or Maven library. The following code snippets show you the required imports to make Retrofit 1 work with OkHttp 3.

Gradle

```
1  compile 'com.squareup.retrofit:retrofit:1.9.0'
2  compile 'com.squareup.okhttp3:okhttp:3.0.1'
3  compile 'com.jakewharton.retrofit:retrofit1-okhttp3-client:1.0.2'
```

Maven

[35] https://github.com/JakeWharton/retrofit1-okhttp3-client

```
<dependency>
  <groupId>com.squareup.retrofit</groupId>
  <artifactId>retrofit</artifactId>
  <version>1.9.0</version>
</dependency>
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
  <version>3.0.1</version>
</dependency>
<dependency>
  <groupId>com.jakewharton.retrofit</groupId>
  <artifactId>retrofit1-okhttp3-client</artifactId>
  <version>1.0.2</version>
</dependency>
```

Use OkHttp 3 Client in RestAdapter

After adding the dependencies to your `build.gradle` or `pom.xml` file and finishing the synchronization, you're ready to plug in the OkHttp 3 client.

The `setClient()` method on the `RestAdapter.Builder` expects a `Client` implementation. Within OkHttp 2, the implementation was called `OkClient`. For OkHttp 3, Jake Wharton named it `Ok3Client` and that's exactly the only thing to change. We just need to pass an `Ok3Client` object as the parameter for the `setClient` method.

```
RestAdapter.Builder builder = new RestAdapter.Builder()
        .setClient(new Ok3Client(new OkHttpClient()))
        .setEndpoint(API_BASE_URL);
```

Keep an eye on the `OkHttpClient` and make sure you're using the OkHttp 3 version. Because OkHttp introduces the new group id called `com.squareup.okhttp3` you're able to have OkHttp 2 and 3 within your project. The `Ok3Client` implements the required `Client` interface, which means there's nothing more to do than enjoy the new features and improvements of OkHttp 3.

OkHttp 3 Advantages and Features

With the update to OkHttp 3, you're able to pick up and use the functionality like powerful interceptors. The thing you need to do is using your `OkHttpClient` for request and response interception instead of Retrofit's interceptor.

The following code block outlines how to use an OkHttp 3 client with added interceptors and an `Authenticator`. The advantage in regard to Retrofit's interceptor is, that you can add multiple of them and not just one.

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

// add interceptors
httpBuilder.addInterceptor(/** Your Interceptor **/)
httpBuilder.addNetworkInterceptor(/** Your Network Interceptor **/)
httpBuilder.authenticator(/** Your Authenticator **/)

// create the OkHttpClient instance from httpBuilder
OkHttpClient client = httpBuilder.build();

RestAdapter.Builder builder = new RestAdapter.Builder()
        .setClient(new Ok3Client(client))
        .setEndpoint(API_BASE_URL);
```

Feel free to explore the greatness of OkHttp 3 interceptors. They enable a lot dimensions and optimizations for your existing project. We can guarantee, you'll love them :)

Release Preparation using OkHttp 3

You're already familiar with the release preparation of your app using Retrofit and OkHttp. When replacing OkHttp v2 with v3, you also need to adjust the proguard rule to skip any warnings regarding v3. Due to the changed group id, update the

from

```
-dontwarn com.squareup.okhttp.*
```

to

```
-dontwarn com.okhttp3.**
```

That's all. You're ready to generate the new APK file with OkHttp enabled.

Wrap-Up

We hope that helps to understand how adding OkHttp 3 to your Retrofit 1 project can improve your project's quality and enables new powerful possibilites (OkHttp 3 interceptors are really powerful!) with the advantages of OkHttp 3.

Outro

Our goal is to truly help you getting started and enlarge upon 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. We think 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.

This book based on Retrofit 1.9.0. The development team works hard towards 2.0 and at the time of this book being written there are only 4 open issues left within the 2.0 milestone. Weâ€™ll update the contents of this book to Retrofit 2.0 as soon as the new release is available. Nevertheless, it will take some time to integrate all the changes. The commit stream points to the fact that multiple classes and methods within Retrofit are renamed. Additionally, some functionality will change and requires us to rewrite some parts of the book and code examples. In short: it will take some time to have the update in place and shipped. However, everyone who bought the book will get the book update **for free**.

In the meantime, feel free to visit our homepage[36] and blog[37]. Weâ€™re publishing new valuable posts every monday and thursday about Android, Node.js and various open source tools.

Thanks a lot for reading this book! We highly appreciate your interest and hope you learned a lot from this book! <3

[36] https://futurestud.io
[37] https://futurestud.io/blog

About the Book

This book is based on the Retrofit series[38] 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[39]
- Email us: info@futurestud.io[40]
- Our blog provides valuable articles: futurestud.io/blog[41]

[38] https://futurestud.io/blog/retrofit-series-round-up/
[39] https://twitter.com/futurestud_io
[40] mailto:info@futurestud.io
[41] https://futurestud.io/blog