Introduction

Retrofit turns your HTTP API into a Java interface.

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

The `Retrofit` class generates an implementation of the `GitHubService` interface.

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

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

Each `Call` from the created `GitHubService` can make a synchronous or asynchronous HTTP request to the remote webserver.

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

Use annotations to describe the HTTP request:

- URL parameter replacement and query parameter support
- Object conversion to request body (e.g., JSON, protocol buffers)
- Multipart request body and file upload

API Declaration

Annotations on the interface methods and its parameters indicate how a request will be handled.

REQUEST METHOD

Every method must have an HTTP annotation that provides the request method and relative URL. There are eight built-in annotations: `HTTP` , `GET` , `POST` , `PUT` , `PATCH` , `DELETE` , `OPTIONS` and `HEAD` . The relative URL of the resource is specified in the annotation.

```
@GET("users/list")
```

You can also specify query parameters in the URL.

```
@GET("users/list?sort=desc")
```

URL MANIPULATION

A request URL can be updated dynamically using replacement blocks and parameters on the method. A replacement block is an alphanumeric string surrounded by `{` and `}`. A corresponding parameter must be annotated with `@Path` using the same string.

```
@GET("group/{id}/users")
Call<List<User>> groupList(@Path("id") int groupId);
```

Query parameters can also be added.

```
@GET("group/{id}/users")
Call<List<User>> groupList(@Path("id") int groupId, @Query("sort") String sort);
```

For complex query parameter combinations a `Map` can be used.

```
@GET("group/{id}/users")
Call<List<User>> groupList(@Path("id") int groupId, @QueryMap Map<String, String> options);
```

REQUEST BODY

An object can be specified for use as an HTTP request body with the `@Body` annotation.

```
@POST("users/new")
Call<User> createUser(@Body User user);
```

The object will also be converted using a converter specified on the `Retrofit` instance. If no converter is added, only `RequestBody` can be used.

FORM ENCODED AND MULTIPART

Methods can also be declared to send form-encoded and multipart data.

Form-encoded data is sent when `@FormUrlEncoded` is present on the method. Each key-value pair is annotated with `@Field` containing the name and the object providing the value.

```
@FormUrlEncoded
@POST("user/edit")
Call<User> updateUser(@Field("first_name") String first, @Field("last_name") String last);
```

Multipart requests are used when `@Multipart` is present on the method. Parts are declared using the `@Part` annotation.

```
@Multipart
@PUT("user/photo")
Call<User> updateUser(@Part("photo") RequestBody photo, @Part("description") RequestBody descri
ption);
```

Multipart parts use one of `Retrofit` 's converters or they can implement `RequestBody` to handle their own serialization.

HEADER MANIPULATION

You can set static headers for a method using the `@Headers` annotation.

```
@Headers("Cache-Control: max-age=640000")
@GET("widget/list")
Call<List<Widget>> widgetList();
```

```
@Headers({
    "Accept: application/vnd.github.v3.full+json",
    "User-Agent: Retrofit-Sample-App"
})
@GET("users/{username}")
Call<User> getUser(@Path("username") String username);
```

Note that headers do not overwrite each other. All headers with the same name will be included in the request.

A request Header can be updated dynamically using the `@Header` annotation. A corresponding parameter must be provided to the `@Header` . If the value is null, the header will be omitted. Otherwise, `toString` will be called on the value, and the result used.

```
@GET("user")
Call<User> getUser(@Header("Authorization") String authorization)
```

Similar to query parameters, for complex header combinations, a `Map` can be used.

```
@GET("user")
Call<User> getUser(@HeaderMap Map<String, String> headers)
```

Headers that need to be added to every request can be specified using an OkHttp interceptor.

SYNCHRONOUS VS. ASYNCHRONOUS

`Call` instances can be executed either synchronously or asynchronously. Each instance can only be used once, but calling `clone()` will create a new instance that can be used.

On Android, callbacks will be executed on the main thread. On the JVM, callbacks will happen on the same thread that executed the HTTP request.

Retrofit Configuration

`Retrofit` is the class through which your API interfaces are turned into callable objects. By default, Retrofit will give you sane defaults for your platform but it allows for customization.

CONVERTERS

By default, Retrofit can only deserialize HTTP bodies into OkHttp's `ResponseBody` type and it can only accept its `RequestBody` type for `@Body`.

Converters can be added to support other types. Six sibling modules adapt popular serialization libraries for your convenience.

- Gson: `com.squareup.retrofit2:converter-gson`
- Jackson: `com.squareup.retrofit2:converter-jackson`
- Moshi: `com.squareup.retrofit2:converter-moshi`
- Protobuf: `com.squareup.retrofit2:converter-protobuf`
- Wire: `com.squareup.retrofit2:converter-wire`
- Simple XML: `com.squareup.retrofit2:converter-simplexml`
- JAXB: `com.squareup.retrofit2:converter-jaxb`
- Scalars (primitives, boxed, and String): `com.squareup.retrofit2:converter-scalars`

Here's an example of using the `GsonConverterFactory` class to generate an implementation of the `GitHubService` interface which uses Gson for its deserialization.

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

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

CUSTOM CONVERTERS

If you need to communicate with an API that uses a content-format that Retrofit does not support out of the box (e.g. YAML, txt, custom format) or you wish to use a different library to implement an existing format, you can easily create your own converter. Create a class that extends the `Converter.Factory` class and pass in an instance when building your adapter.