

BACHELOR PAPER

Term paper submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Engineering at the
University of Applied Sciences Technikum Wien - Degree
Program Computer Science

Using GraphQL with Android Applications

By: Florian Mötz

Student Number: 1610257017

Supervisor: Dipl.-Ing. Dr.techn. Thomas Polzer

Vienna, May 22, 2018

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, May 22, 2018

Signature

Kurzfassung

Die meisten mobilen Applikationen laden oder synchronisieren Daten mit einem Backend-Server oder allgemein aus dem Internet. Dies passiert meistens mit Ressourcen-basierten APIs, welche JSON verwenden. Wenn jedoch eine Liste von Objekten mit einer größeren Anzahl an Feldern geladen werden muss, wobei nur wenige Felder der Objekte benötigt werden, entsteht ein unnötiger Datenüberfluss. Dazu kommt es, da der Server nur eine statische Datenstruktur als Rückgabebetyp erkennt. GraphQL ist eine Query-Sprache für Client-Server Architekturen, in welcher der Client in der Anfrage die genauen Felder, welche zurückgegeben werden sollen, angibt. Dadurch kann unnötiger Datentransfer vermieden werden, ohne mehrere Rückgabebetypen zu definieren. Diese Arbeit untersucht Lösungen, um einen GraphQL Client in Android Applikationen mittels Kotlin einzubinden, und es wird eine neue Kotlin-basierte Lösung für typensicheres Entwickeln von GraphQL Anfragen vorgestellt.

Schlagworte: Android, GraphQL, Code-Generation, Annotation Processing, Kotlin, Retrofit

Abstract

Most mobile applications need to communicate with a back-end server or fetch data in general from the internet, mostly by using resource-based APIs using `JSON`. This architecture, however, has its downsides when loading a list of objects with a large number of fields per object. The server always returns the same object structure with all given fields, however in most cases only a few fields per object are needed by the client, which leads to a lot of data overhead. GraphQL is a query language for client-server architectures in which the client can choose the specific fields to be returned in order to eliminate the data overhead. This work will examine solutions to implement a GraphQL client in Android applications using Kotlin, and will furthermore introduce a new Kotlin based library for type-safe query building.

Keywords: Android, GraphQL, Code-Generation, Annotation Processing, Kotlin, Retrofit

Acknowledgements

I would like to express my appreciation to my supervisor Thomas Polzer for his ever present advice as well as his exceptional help and feedback to shape this thesis.

Moreover I would like to thank my colleagues at Tailored Apps who shared their knowledge and helped in any matter.

Last, but not least, I want to thank my family for their constant support during my study, as well as especially my sister for proof-reading this work.

Contents

1	Introduction	1
1.1	GraphQL vs. REST / JSON	2
1.1.1	REST APIs and their Shortcomings	2
1.1.2	GraphQL	3
1.2	Specification	4
1.2.1	Type system	4
1.2.2	Schema definition	7
1.2.3	Queries	7
1.2.4	Mutations	7
1.2.5	Introspection	7
2	Requirements	9
3	Related Work	10
3.1	Java client libraries for application usage	10
3.2	Apollo Android Client Library	10
3.2.1	Schema processing	11
3.2.2	Create queries	11
3.2.3	Implement query calls	12
4	Methods	13
5	Evaluation of the Apollo library	13
6	Implementation	14
6.1	Used Technologies	14
6.1.1	Kotlin DSL	14
6.1.2	Kotlin Poet	16
6.1.3	Annotation Processing	16
6.1.4	OkHttp / Retrofit	17
6.2	Modules	19
6.2.1	Schema	19
6.2.2	Schema parser	20
6.2.3	Code generator	21
6.2.4	Annotation processor	22
6.2.5	Retrofit adapter	23

7 Results	25
8 Conclusion and Future Work	26
Bibliography	27
List of Listings	31
List of Abbreviations	32

1 Introduction

Smartphones have taken a vital spot in everyday life for the last years. In this area, *Apple* with *iOS* and *Google* with *Android* share the market for smartphone operating systems by 99.3 percent [26]. They both provide an open app distribution channel, where any developer can submit applications to, which enables smartphones to become allrounders for nearly any task.

However, today most mobile applications need to stay in contact with a back-end server in order to synchronize data, save states or message communication in general [36]. Furthermore, a lot of mobile applications are a modified version of websites or another system, provide the user with a more native feeling on their device while representing data stored on servers which get fetched by the application and displayed to the user or stored on the device for off-line functionality [43].

The most common protocol used for data exchange between mobile applications and back-end services is a resource based server (often implementing the Representational State Transfer (REST) specification [35]) using JSON as data format [7]. Besides the easy implementation and little overhead, this architecture presents downsides, especially when dealing with lists of objects with a higher number of fields, which introduce a data overhead by fetching data not actually used by the application [27].

GraphQL, a query language specification targeting the communication with mobile platforms, was built to solve this problem. Furthermore it provides a similar environment as REST and JSON architectures [34].

There are open source GraphQL client libraries for Android applications available, as in the Android developer community it is common to open source libraries. However, these libraries have minor downsides, which is why this thesis has come up with an Android GraphQL client library written in and for Kotlin.

As scope for this thesis, the following research questions are defined:

- How does GraphQL differ from resource-based REST architectures?
- How can a GraphQL client be implement in Android using existing libraries, and which up and downsides are introduced by doing so?
- How can a GraphQL client library be implemented for Kotlin, which overcomes the downsides of existing GraphQL client libraries?

1.1 GraphQL vs. REST / JSON

1.1.1 REST APIs and their Shortcomings

Although any connection protocol is supported by mobile networks (which includes all User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) based protocols), the Hypertext Transfer Protocol (HTTP) protocol is widely used when transferring basic data [7]. Furthermore, as data structure within the HTTP protocol, JSON is used, as it provides little overhead and can be marshalled from and to common data objects in most programming languages easily [6]. This architecture is the de-factor standard for communicating between mobile applications and web services, and therefore it is also easy to implement on a lot of platforms and programming languages using different tools and libraries. A basic resource-based server Application Programming Interface (API) (often referred to as *RESTful*) usually provides resources to retrieve a list of a specific object as well as an endpoint to retrieve, modify or create an object of the given resource (usually identified by a unique id) [55]. In such an architecture, at both the server and the client side, the responded object of a given type is represented with the same fields within the list-response as well as the response of a single item. Therefore, only one data model object needs to be defined and implemented in both, the client and server application.

But given that an object defines a larger number of fields, which themselves can be object types with a lot of fields as well, the response of a list of those objects will grow large. Furthermore, if the mobile application while loading lists of data, only needs a small subset of the fields contained in an object to process or display to the user, a lot of data overhead and therefore a poorer performance is introduced by fetching all fields from every object.

If the API design is changed in favor of only responding with minimized objects at list requests, but the whole detailed object at single object requests, both the server and client are required to implement multiple response classes to map fields for actually the same type of object.

Example

Given an object `Person` which defines the fields `id`, `name`, `birthdate` and `biography`, a JSON response could be defined as in Listing 1. This very simple and lightweight object, without white-spaces or line breaks, counts 91 characters. Given that a request will respond with 100 `Person` objects, the whole response (including the JSON characters for an array definition) will have a payload of $91 * 100 + 2 + 99 * 1 = 9201$ characters.

```
1  {"id":1,"name":"Max Mustermann","birthdate":"1995-05-21","biography":"This is my  
   Profile."}
```

Listing 1: JSON object of a sample Person object

However, if the client only needs to display a list of those objects showing the `name` field and requires the `id` field for the detail call, the object from Listing 1 will be reduced to 33 characters, as seen in Listing 2. A payload containing 100 of those objects, will only contain $33 * 100 + 2 + 99 * 1 = 3401$ characters, resulting in a decrease of 63 percent of payload transmitted.

```
1 {"id":1,"name":"Max Mustermann"}
```

Listing 2: Minified JSON object of a sample Person object

1.1.2 GraphQL

GraphQL is a client-server query language, which was introduced by Facebook and open sourced in 2015 [34]. The basic distinction to resource-based APIs like REST is that in GraphQL the client needs to send a query defining the set of fields to return from the requested object, whereas in REST architectures the client only specifies which resource to retrieve, but not in which granularity [15] [55].

Furthermore, in GraphQL a strongly typed schema is defined and shared between client and server application, which provides libraries and applications to verify queries and responses for validity. In REST and JSON architectures this is often either completely missing or not standardized [16].

In Listing 3 a sample GraphQL query is shown, which requests a `students` query endpoint with a parameter `bornIn` set to 1994, and requests the response to include the name, birthdate and certificates from each student. Within the certificates, only the name and the Uniform Resource Locator (URL) of each certificate should be returned. A sample response to this query is shown in Listing 4.

```
1 query {  
2   students(bornIn: 1994) {  
3     name  
4     birthdate  
5     certificates {  
6       name  
7       url  
8     }  
9   }  
10 }
```

Listing 3: Example GraphQL Query

```

1 {
2   "data": {
3     "students": [
4       {
5         "name": "Max Mustermann",
6         "birthdate": "1994-05-02",
7         "certificates": [
8           {
9             "name": "First Term",
10            "url": "http://example.org/first_term.pdf"
11          }
12        ]
13      }
14    ]
15  }
16 }

```

Listing 4: Example GraphQL Response

1.2 Specification

The GraphQL specification is open sourced at the [Github](#) account of Facebook, where everyone can contribute [14]. Besides the specification and a summarizing *Getting Started* page, Facebook moreover provides a website with step-by-step learning articles with sample code, a blog and a lot of resources for implementing GraphQL architectures [13].

In this section, the basic specification of GraphQL will be covered. They are needed to understand the basic principle of GraphQL as well as interacting with a GraphQL server using basic queries.

1.2.1 Type system

GraphQL distinguishes several different types of definitions, of which the most important ones are described in this section. Any response object or parameter in a query or mutation needs to be one of the defined types [15].

Scalars define a primitive value, such as a string or an integer. GraphQL defines a list of predefined scalars, but within a schema, additional types can be added as scalar types [10]. The following list describes the built-in types [16]:

- `Int` defines a 32 bit signed integer

- `Float` defines a signed floating-point value with doubled precision
- `String` defines a UTF-8 encoded character sequence
- `Boolean` defines a single boolean flag
- `ID` defines a unique identifier, which is represented as string, however fields holding a non-human readable unique identifier should be marked as `ID` instead of a `String`.

To extend the applications schema with another scalar type, only the name of the scalar needs to be defined in the schema, as seen in Listing 5 for the scalar type `Date` [16]. Due to the fact that the schema definition does not contain any information about the representation or serialization of the defined scalar type, the client and server need to define the specification of the scalar type as well as implement converter for the scalar individually [40].

```
1 scalar Date
```

Listing 5: Example of a custom scalar type definition

Object defines a list of named fields with a defined type [10]. Within an object, field names must be unique. This type may be interpreted as a `data class` type in `Kotlin` or any `JSON` object, as they can store the same structure of data. Furthermore, the object name as well as the field names must not start with two underscores, as this prefix is reserved for e.g. the *introspection system* (see Section 1.2.5) [10]. Moreover, object types are allowed to take arguments, which are of any given type and may define a default value [16].

An example object type `Person` can be defined as shown in Listing 6, where a field `name` with the built-in scalar type `String`, the field `age` with the built-in scalar type `Int` and a field `Picture` with another object type (which needs to be defined in the schema) with the argument `resolution` of the scalar type `Int` with a default value of `500` are defined.

```
1 type Person {
2   name: String
3   age: Int
4   picture(resolution: Int = 500): Url
5 }
```

Listing 6: Example of an object type definition

Enums define a type with a given, complete list of possible values [10], also described as a

“special kind of scalar that is restricted to a particular set of allowed values” - [16].

This type is similar to the Java or Kotlin `enum` class. A definition of an enum type specifies the name of the type as well as all possible values, as shown in Listing 7.

```
1 enum Direction {
2     NORTH
3     EAST
4     SOUTH
5     WEST
6 }
```

Listing 7: Example of an enum type definition

Lists and Nullability: All given types may also be present as a list of values, which is indicated by surrounding the type name with square brackets [10]. This list of types may always contain zero or more elements. Furthermore, any field is *nullable* by default, adding an exclamation mark at the end of the type declares the field as mandatory or non-nullable. Listing 8 shows example definitions of lists and *non-nullable* types. The field `name` is marked as mandatory and therefore may never be `null`. The field `friends` is a list containing values of the object type `Person`. Adding an exclamation mark at the end of the type name within the brackets indicates that the field `friends` may be `null`. However, if the field is not `null`, the list may never contain values which are `null`. The field `interests` has the exclamation mark set outside of the brackets, which indicates that the field `interests` may never be `null`, but elements within the list may be `null`.

```
1 type Person {
2     name: String!
3     friends: [Person!]
4     interests: [String]!
5 }
```

Listing 8: Example of lists and non-nullable types

Query and Mutation are reserved type names, which may be specified as object types (see Object in Section 1.2.1) in the schema to declare possible queries and mutations (see Section 1.2.3 and 1.2.4) [10]. A query defines a request with optional parameters, which return a defined type. A mutations is defined similarly, but is allowed to apply changes to the data on the server. As an example, Listing 9 declares a query and mutation type.

```

1 type Query {
2   me: Person
3   searchPerson(name: String!): [Person!]
4 }
5 type Mutation {
6   addInterest(name: String!, interest: String!): Person
7 }

```

Listing 9: Example of a Query and Mutation type definition

1.2.2 Schema definition

GraphQL specifies an Interface Definition Language (IDL) to describe the schema including all types, query and mutation definitions [16]. This schema can be shared between server and client to provide a well defined, single source of truth for the strongly typed system, which makes communication or definition errors during development and query execution less error-prone.

1.2.3 Queries

As GraphQL is a query language, queries need to be defined in a schema as well as any other type [9]. Every request to a GraphQL server is either a `query` or `mutation` (see Section 1.2.4) [16]. A query may take arguments and must return a specified type, but as the special feature of GraphQL is to only return fields required by the client, the client needs to specify in a query, which fields of the given return type should actually be returned. At least one field needs to be declared, as running a query without any field declared will result in an error [15]. The implementation of a query needs to be read-only, so no data manipulation is allowed to take place [9]. In a schema, queries are defined within the `Query` object definition as ordinary fields, which may take arguments and have a return type.

1.2.4 Mutations

A mutation can be defined as a query which manipulates data [9]. A mutation may take arguments, and will return a given type. The definition of a mutation in a schema is the same as for `Queries` (see Section 1.2.3), but within the `Mutation` type definition.

1.2.5 Introspection

The GraphQL system provides the so-called *introspection system*, which enables to execute specific, predefined queries, in order to receive information about the available fields and types within a GraphQL schema implemented by a server. [12] [11] [1]. By using GraphQL queries e.g. on the predefined type `__type`, the kind and name of the type specified by the given

name can be retrieved as well as all fields defined by the given type. This enables the client to check for the specified types or queries using the same query system as standard queries use (see Section 1.2.3). An example introspection query is shown in Listing 10, an example response of this introspection query is shown in Listing 11.

```
1 query {
2   __type(name: "Person") {
3     kind
4     name
5     fields {
6       name
7     }
8   }
9 }
```

Listing 10: Example of an Introspection Query

```
1 {
2   "data": {
3     "__type": {
4       "kind": "OBJECT",
5       "name": "Person",
6       "fields": [
7         {
8           "name": "name"
9         },
10        {
11          "name": "age"
12        }
13      ]
14    }
15  }
16 }
```

Listing 11: Example of an Introspection Query Response

2 Requirements

After introducing the `GraphQL` specification in the last section, the requirements set for a `GraphQL` client library developed in `Kotlin` will be defined. As `GraphQL` is strongly typed, developing and writing queries should also be type-safe and easy for developers. Therefore, a client library should provide auto-completion for type fields, error-detection as well as type safety acquired from the given schema. The evaluated must-haves for such a library is presented in the following list:

- The library needs to be able to parse local *GraphQL Schema Files* for further processing and to create queries with the types defined in the given schema.
- The library is able to let the developer create a `GraphQL` schema using a `Kotlin` Domain Specific Language (DSL).
- Using the schema created by one of the methods above, the library needs to be able to build queries within a type-safe builder function, including auto-completion.
- Furthermore, the library implements an easy way to serialize the created queries into a query string, which can be sent to a server. Moreover, the library provides an easy to use interface for developers, or provides integrations for common Android programming frameworks, like `Retrofit` or `RxJava`.
- The library needs to generate response classes for the types available in the schema. In this way, the library will create a strongly typed environment for the developer. Furthermore, these classes can be used to de-serialize a response from a query into a response class. Moreover, also the de-serialization should be automated for the developer, by providing either an easy to use interface, or integrations for common Android programming frameworks, like `Retrofit`.

3 Related Work

This section will introduce available libraries to implement a GraphQL client within Android applications.

3.1 Java client libraries for application usage

There exist numerous libraries for GraphQL server and client applications in different programming languages and platforms, and also a lot of libraries in Java or Kotlin [5]. However, only three open source libraries to be implemented in Android applications have been found, as a lot of the libraries are either server libraries, or bound to a specific framework. [5]:

- `graphql-java` is a Java client for GraphQL queries, but as it is not intended for Android implementation, this library has not been evaluated, because it implements a custom network client, which is uncommon in Android applications [4].
- `LiveGQL` is a Java client library to execute GraphQL queries, but does not provide any schema handling or type-safe builder. Furthermore, it uses `Tyrus` internally for managing websockets, which is not favored in Android development [3].
- `apollo-android` is the Android library of the Apollo framework, which provides a large number of GraphQL libraries for both servers and clients on different platforms.

Nevertheless, the *Apollo Android* library has been identified to be suitable for Android applications and at least a large subset of the defined requirements (see Section 2).

3.2 Apollo Android Client Library

The most advanced client library for Android at the moment is the *Apollo GraphQL Client for Android* with nearly 900 stars on Github [40]. It is the Android library of the Apollo-framework, which offers open source GraphQL libraries for a lot of client and server platforms and programming languages [39] [37].

This library includes a code generator, which, given a preprocessed schema (see Section 3.2.1) and queries stored as `.graphql` files in the project, creates builder and response classes in Java for the given queries. Those queries can be sent to the configured GraphQL server using the library with an asynchronous callback, where internally the well-established

`OkHttp` library is used for the HTTP connection handling. Furthermore, the library provides modules for `RxJava` support for query handling, as well as a *HTTP cache* for caching query operations [40].

3.2.1 Schema processing

The Apollo Android Client Library needs a preprocessed schema at compilation which is used to generate the required classes for query builder methods [40]. For this pre-processing, the Apollo framework provides the Node Package Manager (npm) package `apollo-codegen`, which requires an URL as parameter to the GraphQL server which implements the wanted specification and produces a proprietary JavaScript Object Notation (JSON) file representing the schema of the given GraphQL server using the *introspection system* (see Section 1.2.5) [40] [38]. This JSON file needs to be placed in the applications source set within the same folder as the queries created in Section 3.2.2.

3.2.2 Create queries

To create queries (or mutations), the queries need to be defined in `.graphql` files in order to let the Apollo library create Java classes during the build process, which can be used as builder-methods to create the queries in the application [40]. The queries need to be placed in the `sourceSet graphql` in the *main* module of the Android application. Within that `sourceSet`, the folder structure, in which the query file is placed, defines the package the classes will be generated in [40].

In order to generate the query classes in the package `com.example.graphql.api`, the query file needs to be stored in the following location:

```
src/main/graphql/com/example/graphql/api/ExampleFeedQuery.graphql
```

The file contains standard query definitions for each query the application needs to perform, whereas the schema types do not need to be defined, as they are represented in the generated JSON file (see Section 3.2.1).

An example query which can be inserted in the `ExampleFeedQuery.graphql` file is shown in Listing 12.

```
1 query SearchQuery($search: String!) {  
2   searchPerson(name: $search) {  
3     name  
4   }  
5 }
```

Listing 12: `ExampleFeedQuery.graphql`

3.2.3 Implement query calls

To use the generated classes to actually query the GraphQL server in the application, the Apollo Android library provides a `ApolloClient` class [40], on which connection configurations can be set. The library internally uses `OkHttp`, which is a commonly known and used framework for network communication in Android application, that allows a lot of different configurations. Once an `ApolloClient` instance has been created, the object provides a `query` method, which can be used to send queries. The library has generated builder classes for every query defined in the `graphql` files in Section 3.2.2. The usage of such builder classes is shown in Listing 13.

```
1  apolloClient.query(  
2      SearchQuery.builder()  
3          .search("Max")  
4          .build()  
5  ).enqueue(new ApolloCall.Callback<SearchQuery.Data>() {  
6  
7      @Override public void onResponse(@NonNull Response<SearchQuery.Data>  
8          dataResponse) {  
9          final StringBuffer buffer = new StringBuffer();  
10         for (SearchQuery.Data.Person person : dataResponse.data().person()) {  
11             //person.name() retrieves the name of the person  
12         }  
13  
14         @Override public void onFailure(@NonNull Throwable t) {  
15             //An error occurred  
16         }  
17     });
```

Listing 13: Sample Apollo Client usage

4 Methods

To evaluate ways to implement a GraphQL client within an Android application, at first publicly available libraries were searched and evaluated. The evaluation of these libraries has been done by checking the applicability within Android applications (in terms of network libraries used, e.g.) or the provided functionality as well as the easiness of implementation.

For the implementation of a custom Android library, the requirements set in Section 2 have been used to architect the functionalities of the library. These requirements have been evaluated after analysing the needs for a GraphQL android library for developers, if no running GraphQL server is available during development. Furthermore, the requirements are chosen based on the common android libraries already used by the Android developer team in the company.

5 Evaluation of the Apollo library

To evaluate how to implement a GraphQL client in Android applications best, the currently best evolved and most established open source library (see Section 3.1) was taken into consideration and implemented in a sample application. Within the implementation of the sample application, the suitability of the library regarding the requirements set in Section 2 was tested. The outcome of the library evaluation is shown in the following list:

- The library needs a running and completely implemented GraphQL server for the build process to succeed. This is due to the fact, that the library needs a separately created `schema.json` file in order to build the required classes. The `schema.json` file is generated using a separately provided `npm` package, as described in Section 3.2.1, which needs a server to be already running. The library will fail the build if it cannot find the generated `schema.json` within the project. Furthermore, the generation of the `schema.json` file is not automated nor implemented in the build process, so if schema changes occur, the developer needs to manually regenerate this file.
- The library does not provide auto-completion nor type-safety when creating the queries in the `.graphql` files (see Section 3.2.2). This leads to an increased effort in looking up field types in the schema specification. Furthermore, as no auto-completion is provided,

typing errors are harder to find and only get signalled to the annotation processor when building the application.

- The library does not allow to create schemas within the application or the build process. The schema is only generated by querying a GraphQL server using the `introspect` call (see Sections 1.2.5 and 3.2.1).

Due to this reasons, the Apollo GraphQL Android library was disqualified as a suitable solution for a GraphQL android client library.

6 Implementation

Due to the reasons stated in Section 5, the most evolved Android client library, Apollo Android, did not meet the demands set by our company. Therefore, the development of a GraphQL client library for Android applications has been initiated. The goals of this library are to meet all requirements listed in Section 2.

The library, however, does not need to implement the full GraphQL specification, as queries and mutations with any types but custom scalar types are sufficient for most usages. Furthermore, the library should be developed using Kotlin and should be targeting Kotlin applications.

6.1 Used Technologies

In this section, the most important technologies used to develop the GraphQL library are described. The project generally is developed using Kotlin instead of Java, as it addresses and improves a series of downsides of the Java programming language, such as null-safety, smart casts or extension functions [28]. Furthermore, Google features Kotlin as “an official language on Android” [20] since *Google I/O 2017*.

6.1.1 Kotlin DSL

To create a Kotlin DSL, as done to create a type-safe builder, the following two Kotlin features have been used.

Lambda Method with a receiver

Kotlin allows lambda methods to define a receiver type, also called “*function type with receiver*” - [30], which will be available implicitly as `this` within the lambda function body [30].

Using that, developers can call methods within a lambda method, which are invoked automatically on an object internally. A simple sample is shown in Listing 14. Within this listing, the method `greet` requires a lambda as parameter, which defines no parameters and returns `Unit`, which is the Kotlin equivalent of `void`. However, within the body of the lambda passed to the `greet` method, the `this` object is implicitly of the type `ExampleReceiver`, which enables the lambda body to call the `hello` method of the given type, as seen in Listing 15. The object, which is available as `this` within the lambda body, needs to be passed to the `invoke`-method of the lambda, as seen within the `greet` method in Listing 14.

```
1 class ExampleReceiver {
2     fun hello(name: String) {
3         println("Hello, $name!")
4     }
5 }
6
7 fun greet(lambda: ExampleReceiver.() -> Unit) {
8     val receiver = ExampleReceiver()
9     lambda.invoke(receiver)
10 }
```

Listing 14: Sample Kotlin lambda receiver

```
1 greet(lambda = {
2     hello("World")
3 })
```

Listing 15: Usage of the lambda receiver

Methods with lambda parameters as last parameter

In Kotlin, parameters may be passed anonymously or by explicitly setting names [30]. In both cases, however, if the last parameter of a method definition is a lambda function, the brackets may be closed before defining the lambda parameter using curly brackets. If no other argument is passed, or the lambda is the only parameter defined, the brackets may be omitted completely, as seen in Listing 16, which represents the same functionality as Listing 15.

```
1 greet {
2     hello("World")
3 }
```

Listing 16: Example of a lambda parameter as single parameter and omitted brackets

6.1.2 Kotlin Poet

For the process of generating `Kotlin` code, the files could be created by manually writing each statement and character into the file stream. However, this process would be very error-prone and unmaintainable, for which *Square* has open sourced a `Kotlin` code generation framework called `KotlinPoet` [54]. This library provides builder classes for any kind of object to generate in `Kotlin`-files.

Listing 17 shows an example usage of the `Kotlin Poet` library to generate a `HelloWorld.kt` file with a class `Sample` defining a method `helloWorld`. The file object, which is returned by the `FileSpec` builder method then provides `writeTo` methods to write the generated file into any file or stream.

```
1 val file = FileSpec.builder("", "HelloWorld")
2   .addType(TypeSpec.classBuilder("Sample")
3     .addFunction(FunSpec.builder("helloWorld")
4       .addStatement("System.out.println(%S)", "Hello, World!")
5       .build())
6     .build())
7   .build()
```

Listing 17: Sample Kotlin Poet Code Generation

6.1.3 Annotation Processing

To generate code within the build process of an application, *Annotation Processing* can be used. Annotation Processing is the processing of the developed code at compile time, more precisely just before the code is compiled using the `Java` or `Kotlin` compiler [25]. This means, that within annotation processing, code can be generated, which will be included into the compilation of the application.

By using the `kotlin-android` gradle plugin, default build chains and tasks are created automatically [31].

As the code and the application are written in `Kotlin`, also the Kotlin Annotation Processor (`KAPT`) instead of the standard Java Annotation Processor (`APT`) needs to be used. This is due to the fact that in the gradle build chain, the `APT` takes place after the compilation of the `Kotlin` classes, where `Kotlin` files generated by the `APT` would not be included within the compilation. The build process roughly can be defined in the following sequence:

- `generateStubs`: This task generates stubs for all `Kotlin` classes, which declare the same method signatures and fields, but do not contain bodies of methods [2].
- `kaptKotlin`: This task invokes the `KAPT` with the generated stubs [32].
- `compileKotlin`: This task compiles the original `Kotlin` classes combined with the generated classes from the `kaptKotlin` task.

- `apt`: This task would be the Java annotation processor, but if the project uses Kotlin code, any annotation processing should be performed using the `KAPT`, as the Kotlin compiler, which is run before the `apt` task, is not able to include classes generated by the Java annotation processor.
- `compileJava`: This task invokes the Java compiler with the Java source files as well as the class-files generated by the Kotlin compiler.

The Kotlin Annotation Processor then, however, internally uses the `javac` compiler to invoke the annotation processors, which means that the standard Java Annotation Processor interface classes can be used to implement annotation processing in Kotlin, even though the annotation processor itself is written in Kotlin [32].

To implement a simple annotation processor, the Java framework provides an `AbstractProcessor`, which can be implemented, as seen in Listing 18. The subclass should override the following methods [41]:

- `getSupportedAnnotationTypes`: Should return a `MutableSet` of annotation classes which the given annotation processor is going to process. Only if annotations returned by this method are found within the source code, the `process` method will be invoked eventually.
- `getSupportedSourceVersion`: Should specify which source code version the processor is able to process (e.g. Java 8).
- `process`: the method which will be invoked to process the annotations found in the source-code.

Moreover, annotation processors need to specify their processors in the `META-INF` meta-data folder, defining the location of the processor (package name and class name). Since this task is often forgotten by the developer, Google has open sourced the `AutoService` library, which automates this process by annotating the processor with the `@AutoService` annotation, where only the `Processor` class, which internally will invoke the annotation processor, needs to be passed as argument [24].

Within the library, the annotation processor could only be made functional by including an annotation within the source code, which is returned by the `getSupportedAnnotationTypes` method. Therefore, an annotation class has been implemented and needs to be present at least once at any class in the source code.

6.1.4 OkHttp / Retrofit

`OkHttp` is an open source library to implement `HTTP` clients for Android (or generally, Java) applications, developed and maintained by *Square* [44]. Although providing an easy to use

```

1 @AutoService(Processor::class)
2 class SampleProcessor : AbstractProcessor() {
3
4     override fun getSupportedAnnotationTypes(): MutableSet<String> {
5         return mutableSetOf(AnnotationToProcess::class.java.name)
6     }
7
8     override fun getSupportedSourceVersion(): SourceVersion {
9         return SourceVersion.latest()
10    }
11
12    override fun process(set: MutableSet<out TypeElement>, roundEnv:
        RoundEnvironment): Boolean {
13        // Here the processing should take place.
14        // Returning true indicates success
15        return true
16    }
17
18 }

```

Listing 18: Sample Annotation Processor construct using Kotlin and the AutoService annotation

interface for programmers with builder methods, the library also provides a high level of configuration, for example for HTTPS configuration, call interceptors or caching [51] [49] [50]. Listing 19 shows a minimal example to send a network request using `OkHttp`. Alternatives to `OkHttp` and `Retrofit` are, for example, to use the `URLConnection` provided by the Android platform, which, however, provides a more complex API interface as well as does not handle threading and needs the developer to write a lot of boilerplate code [17]. Furthermore, Google also provides the open source library `Volley`, which offers an API with less boilerplate code to write for the developer, but is not as popular as `OkHttp` (about 1500 times starred on Github [23]) and still requires more boilerplate code than `Retrofit` [21].

```

1 //A client can be shared for multiple requests
2 val client = OkHttpClient()
3
4 val request = Request.Builder()
5     .url("http://example.org/")
6     .build()
7
8 val response = client.newCall(request).execute()
9
10 val responseString = response.body().string()

```

Listing 19: Sample HTTP request using `OkHttp`

`Retrofit` is an open source library, also developed by *Square*, which enhances the basic

`OkHttp` as a type-safe, simplified implementation of HTTP requests by only defining an annotated interface and the request / response parameters [48]. `Retrofit` uses reflection as well as adapters and converters to create `OkHttp` requests from a given interface and for example adapt the call using reactive programming semantics using `RxJava` [42].

Both libraries are well established and used a lot in the Android developer community, and are seen as the *de-facto* standard for network communication in Android by a lot of Android developers. This is also affirmed by the fact that both libraries are starred more than 25.000 times on Github each, as well as that they are used by official guides of the android developer team [45] [48] [19].

6.2 Modules

The library has been split into multiple `modules` (which are respectively `Gradle` modules), as each of the modules are standalone parts, which may depend on each other, and are required in different stages of the build process. For example, the annotation processor and code generator only need to be present at compile-time within the annotation processing, while the schema DSL and generated classes need to be compiled and present at runtime. In this section, all developed modules are explained in terms of functionality, objective and possible drawbacks of the selected solution.

6.2.1 Schema

This module implements the `GraphQLSchema` class, which provides the `Kotlin` DSL to create a schema using `Kotlin`. Furthermore, this class is used within the library to specify a schema at for example the schema parsing or code-generation.

For the developer, this class provides the type-safe builder (as described in Section 6.1.1) for generating schemas. Therefore, the following methods are available within the type-safe builder:

- `type` defines a new object type (see Section 1.2.1). This method takes as parameters the name of the type as string, as well as another type-safe builder with the type `GraphQLType` as receiver, to specify the type, which provides the following methods:
 - `element` defines a new element, which is defined by the method parameters as well as the `TypeSpecification` type-safe builder function passed.
 - `elementArray` defines a new array of elements, which is defined by the method parameters as well as the `TypeSpecification` type-safe builder function passed.
 - `implements` defines the name of an interface this object type is implementing.
- `query` defines the query type (see Section 1.2.3), and provides the same characteristics as the `type` method mentioned above for query-types.

- `mutation` defines the mutation type (see Section 1.2.4), and provides the same characteristics as the `type` method above for mutation-types.
- `enum` defines a new enum type (see Section 1.2.1), and provides a type-safe builder to add values to the `enum` as lambda method.
- `scalar` defines a new scalar type (see Section 1.2.1) with the given name.

In Listing 20, an example of using the type-safe builder to create a schema is shown.

```

1  val schema = GraphQLSchema.createSchema("Sample") {
2      type("Person") {
3          element(name = "name", mandatory = true, type = ScalarType.String)
4          element(name = "age", mandatory = false, type = ScalarType.Int)
5          element(name = "picture", mandatory = false, type = "Url") {
6              argument("resolution", ScalarType.Int, 500)
7          }
8      }
9
10     type("Url") {
11         element(name = "url", mandatory = true, type = ScalarType.String)
12         element(name = "cacheTime", mandatory = true, type = ScalarType.Int)
13     }
14
15     enum("Direction") {
16         value("NORTH")
17         value("EAST")
18         value("SOUTH")
19         value("WEST")
20     }
21
22     query {
23         element("me", type = "Person")
24         elementArray("searchPerson", type = ScalarType.String) {
25             argument("name", ScalarType.String)
26         }
27     }
28 }

```

Listing 20: Creating a schema using the Kotlin DSL

6.2.2 Schema parser

The schema parser module is intended to parse a given schema string or file into a schema object provided by the schema module (see Section 6.2.1). To improve the easiness of implementation, however, the implemented schema parser only supports basic schema types, such as object types, queries, mutations, enums and scalars. Moreover, only a specific formation of

the schema file is supported, for example only one definition or enum value per line. However, the schema parser meets the basic requirements for the library for the implemented schema types.

Within the implementation, and especially since not the full GraphQL specification is implemented, but only object, enum and scalar types in a special formatted way, the parser is designed to print detailed exceptions if an error occurs. The exceptions include the cause, a possible correct example of the parsed file, the line number of the error and the content of that line to provide as much debugging information to the developer as possible.

The schema parser makes strong usage of the `Kotlin stdlib String Extension Functions`. Kotlin architecture allows to declare statically resolved *Extension functions* on Types to add functionality without extending the class. Extension functions are resolved statically at compile time, so they do not modify the *extended* class, but implement static methods with the receiving object as parameter [29]. The `Kotlin stdlib` provides useful extension functions, such as `subStringAfter` or `subStringBefore`, which return a substring after or before a given character.

6.2.3 Code generator

Within the code generator module, the classes needed for type-safe query builder, as well as the classes for the types defined in the schema and the response classes are generated. To generate the classes, `KotlinPoet` is used (see Section 6.1.2).

All classes and methods for a schema are implemented within a class with the name derived from the name of the schema-file suffixed by `GraphQL`. For example if the file is named `example.graphql`, the class will be named `ExampleGraphQL`.

At first, all enum type definitions are created, by generating a `enum class` with the given values.

Afterwards, all object type definitions are generated, where for each type a `data class` with the given fields, reflecting the nullability of the field on the `Kotlin` field. In `data classes`, the default constructor needs to implement all given declared fields. Another constructor with zero arguments is created, which calls the primary constructor with `null` as parameters for nullable fields, invoking the *zero-arguments* constructor of an object type if its not nullable, or a default value for scalar types (an empty string for `String` or 0 for integers, e.g.) in order to let the response converter create an *empty* object of the given type.

Moreover, for each query type, a `QueryBuilder` class is generated, which declares methods for the type-safe creation of queries at runtime. Within these `QueryBuilder` classes, the generated response wrapper classes are placed as well.

In the end, the generator adds a `companion object` with a method for each query in order to create the query using a lambda with the generated `QueryBuilder` of the given type, to the schema class. `companion objects` represent a static class in Kotlin, which uses the same name as the containing class, which is the preferred way to add *static methods* within

Kotlin classes.

Listing 21 shows a sample usage of the Kotlin DSL provided by the generated classes, which queries the `me` endpoint and requests the name of the `Person`, as well as the small picture URL to be returned. The object returned by the `build` method provides a `toQueryString` method, which serializes the built query into the request string, which can be sent to the GraphQL server. Listing 22 shows the query string generated by the builder defined in Listing 21.

```
1 val query = SchemaGraphQL.buildMeQuery {
2     name()
3     picture {
4         small()
5     }
6 }
```

Listing 21: Create a Query using a Kotlin DSL provided by generated classes

```
1 query {
2   me {
3     name,
4     picture {
5       small
6     }
7   }
8 }
```

Listing 22: Generated query string from a query created using a Kotlin DSL provided by generated classes (added line indents for better readability)

6.2.4 Annotation processor

Within the annotation processor module, the source set is searched for any file with the `graphql` extension. The files found in the source-set are then passed onto the code generator (see Section 6.2.3). The files generated by the code generator are then saved within the directory for generated code, which can be retrieved by the `kapt.kotlin.generated` variable set within the processing environment [41].

To walk recursively through every sub-directory in a given directory, the Kotlin standard library provides a `walkTopDown` *Extension function* on the `File` type, which returns a `Sequence` of all files found recursively [33]. Applying *filter* on the given `Sequence`, a list of files matching a given extension can be retrieved easily as shown in Listing 23.

Moreover, the `ProcessingEnvironment` provided in the `init` method of the `AbstractProcessor`, which may be overridden, provides a `messenger`, which can be used

```

1 val locations: List<File> = File("./")
2     .walkTopDown()
3     .filter(File::isFile)
4     .filter { it.extension.equals("graphql", ignoreCase = true) }
5     .toList()

```

Listing 23: Example of retrieving a list of all files with the (case-insensitive) extension graphql

to write messages within the build log. As messages printed using the default `System.out` stream are not printed in the build process, this messenger object should be used to print messages of different levels (Error, Warning or Note, e.g.).

6.2.5 Retrofit adapter

The last module of the library is the implementation of `Retrofit Adapters` to enable `Retrofit` to process the generated types as input and output parameters.

`Retrofit` provides two possibilities to change the interface for an API call: *Adapter* and *Converter* [52].

Adapters are used to map calls to a form of asynchronous callback, which is used as return value of the defined interface [52]. For example, `Retrofit` provides an adapter to use `RxJava Observable` or `Completable` as return types for the interface call. An adapter will be called to adapt the `Call` instance provided by the `OkHttp` library used internally into the defined type in the `Retrofit` interface, to provide the given type as interface to the developer.

Converters are used to map the body of a response into a defined type, `Retrofit` provides an `Adapter` to de-serialize a response body using `Gson` into a given data class, e.g. [46] [47].

Since for the use of this library the response body should be mapped, a converter needs to be provided. To do so, a `Converter.Factory` needs to be implemented, which may implement the following methods [53]:

- `responseBodyConverter` returns a converter from a given `ResponseBody` object to the given type, or null if the given type is not handled by the factory.
- `requestBodyConverter` returns a converter from the given type to a `RequestBody` object, if the factory is handling the given type.
- `stringConverter` returns a converter from the given type to a string, if the given type is passed as a field, header, path or query parameter to the retrofit interface.

For the library, a `ResponseBodyConverter` to map the HTTP response into the generated response data classes, and a `StringConverter` to serialize query objects passed into `Retrofit` interface methods as parameters have been implemented.

The developer only needs to add the converter factory provided by the library to the retrofit builder to use the given converters, as shown in Listing 24. The `Retrofit` interface only needs

to implement the HTTP method and path for the query, a query-parameter for the build query with the generated builder-type, and the generated response-type, as shown in Listing 25.

```
1 Retrofit.Builder()
2     .baseUrl("https://example.org/")
3     .addConverterFactory(GraphQLConverterFactory.create())
4     .callFactory(httpClient)
5     .build()
6     .create(RetrofitApiInterface::class.java)
```

Listing 24: Example usage of the GraphQLConverterFactory by building a retrofit instance

```
1 interface RetrofitApiInterface {
2
3     @GET("graph")
4     fun query(
5         @Query("query") query: SchemaGraphQL.MeQueryBuilder
6     ): Single<SchemaGraphQL.MeQueryBuilder.Response>
7
8 }
```

Listing 25: Usage of the Converter in a retrofit interface

7 Results

The evaluation of the possible ways to implement a GraphQL library in Android applications has been performed by implementing the different solutions into a sample application. Besides validating, if the set requirements (see Section 2) are met by each solution, the complexity of implementing the solution has been taken into account.

As a sample application to test the library, the *Github* API has been used as a GraphQL server, as it offers all content available using a public API. From the API, the list of repositories starred by the requesting user has been requested and displayed with various details about the repository, such as the name, link, author or licensing information.

While comparing the Apollo Android library (see Section 3.2) with the one created within this work (see Section 6), the following differences could be determined:

- The Apollo library has implemented all GraphQL specifications and can parse queries containing fragments or union types, which the implemented library is not able to do yet.
- The Apollo library furthermore is able to generate the classes using the introspection system of a running server without a schema definition file, which the implemented library is not able to do yet.
- The implemented library is able to compile without a running server by using a schema definition file to generate the needed files, which the Apollo library is not able to do. Moreover, the Apollo library needs the developer to run an external (provided) tool to generate a proprietary schema file by introspecting the GraphQL server.
- The implemented library is able to generate classes by using a provided schema definition or a schema defined within the application using a DSL, which the Apollo library lacks.
- The Apollo library, however, is part of a frequently used and developed framework family, which means that the library is well-engineered and tested in a lot of applications and use-cases, whereas the implemented library has just been developed and lacks of some functionality.
- The implemented library provides type-safe building of queries, whereas the Apollo library needs the queries in a plain text-file for the build-process, where no auto-completion or type-safety is provided.

8 Conclusion and Future Work

It can be concluded, that the Apollo Android library can be used if the GraphQL server is already up and running, and auto-completion or type-safety while developing queries is not needed. Auto-completion may not be needed, as most GraphQL server provide a user interface (like `GraphiQL`) to create queries, which afterwards can be transferred into the application [8].

However, if the development of the server is not finished before starting the development of the Android application, which is often the case in our company, the Apollo framework is cumbersome to use. In this case, for basic GraphQL queries it is better to use the implemented library, as only a schema definition file is needed to generate the code needed for the application. The downside of using the implemented library, is that not all specifications of GraphQL have been implemented, and it has not been tested in a lot of applications or use cases yet, which may reveal some errors.

In future work, performance differences as well as the impact of method references count with and without using *ProGuard*, which is an important figure within Android development, should be evaluated [18] [22]. Furthermore, the library needs to be extended to implement more GraphQL specifications, such as union types, fragments or subscriptions. Moreover, the use of `Retrofit` converters versus providing a custom, easy to use API interface, using `RxJava` semantics e.g., should be considered.

Bibliography

- [1] ALLIGATOR.IO: *Introspection Queries With GraphQL*. <https://alligator.io/graphql/introspection-queries/>, April 2017. Accessed: 14.05.2018.
- [2] ANDREY BRESLAV: *Better Annotation Processing: Supporting Stubs in kapt*. <https://blog.jetbrains.com/kotlin/2015/06/better-annotation-processing-supporting-stubs-in-kapt/>, June 2015. Accessed: 25.04.2018.
- [3] BILLY BICHON: *billybichon/liveGQL: Simple java library in order to use GraphQL Subscription on WebSocket based on Apollo Protocol..* <https://github.com/billybichon/liveGQL>, 2017. Accessed: 27.04.2018.
- [4] BRAD BAKER: *Welcome to graphql-java — graphql-java current documentation*. <http://graphql-java.readthedocs.io/en/latest/index.html>, 2017. Accessed: 27.04.2018.
- [5] C. T. LIN: *chentsulin/awesome-graphql: Awesome list of GraphQL & Relay*. <https://github.com/chentsulin/awesome-graphql>, April 2018. Accessed: 27.04.2018.
- [6] ECMA INTERNATIONAL: *The JSON Data Interchange Syntax*. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, December 2017. Accessed: 14.05.2018.
- [7] EIEI THU, THAN NWE AUNG: *Developing mobile application framework by using RESTFuL web service with JSON parser*. https://www.researchgate.net/publication/281080973_Developing_mobile_application_framework_by_using_RESTFuL_web_service_with_JSON_parser, August 2015. Accessed: 14.05.2018.
- [8] FACEBOOK, INC.: *graphql/graphiql: An in-browser IDE for exploring GraphQL..* <https://github.com/graphql/graphiql>, August 2015. Accessed: 29.04.2018.
- [9] FACEBOOK INC.: *graphql/Section 2 – Language.md*. <https://github.com/facebook/graphql/blob/master/spec/Section%202%20--%20Language.md>, July 2015. Accessed: 24.04.2018.
- [10] FACEBOOK INC.: *graphql/Section 3 – Type Systems.md*. <https://github.com/facebook/graphql/blob/master/spec/Section%203%20--%20Type%20System.md>, July 2015. Accessed: 24.04.2018.
- [11] FACEBOOK INC.: *graphql/Section 4 – Introspection.md*. <https://github.com/facebook/graphql/blob/master/spec/Section%204%20--%20Introspection.md>, July 2015. Accessed: 27.04.2018.

- [12] FACEBOOK INC.: *Introspection / GraphQL*. <http://graphql.org/learn/introspection>, September 2016. Accessed: 27.04.2018.
- [13] FACEBOOK INC.: *GraphQL / A query language for your API*. <http://graphql.org/>, March 2018. Accessed: 24.04.2018.
- [14] FACEBOOK INC.: *GraphQL is a query language and execution engine tied to any backend service..* <https://github.com/facebook/graphql>, February 2018. Accessed: 24.04.2018.
- [15] FACEBOOK INC.: *Queries and Mutations / GraphQL*. <http://graphql.org/learn/queries>, Februar 2018. Accessed: 24.04.2018.
- [16] FACEBOOK INC.: *Schemas and Types / GraphQL*. <http://graphql.org/learn/schema>, April 2018. Accessed: 24.04.2018.
- [17] GOOGLE DEVELOPERS: *Connect to the network / Android Developers*. <https://developer.android.com/training/basics/network-ops/connecting>, 2017. Accessed: 28.04.2018.
- [18] GOOGLE DEVELOPERS: *Enable Multidex for Apps with Over 64K Methods / Android Developers*. <https://developer.android.com/studio/build/multidex>, 2017. Accessed: 29.04.2018.
- [19] GOOGLE DEVELOPERS: *Guide to App Architecture / Android Developers*. <https://developer.android.com/topic/libraries/architecture/guide>, 2017. Accessed: 26.04.2018.
- [20] GOOGLE DEVELOPERS: *Kotlin and Android / Android Developers*. <https://developer.android.com/kotlin/>, 2017. Accessed: 28.04.2018.
- [21] GOOGLE DEVELOPERS: *Send a simple request / Android Developers*. <https://developer.android.com/training/volley/simple>, 2017. Accessed: 28.04.2018.
- [22] GOOGLE DEVELOPERS: *Shrink Your Code and Resources / Android Developers*. <https://developer.android.com/studio/build/shrink-code>, 2017. Accessed: 29.04.2018.
- [23] GOOGLE, INC.: *google/volley*. <https://github.com/google/volley>, 2013. Accessed: 28.04.2018.
- [24] GOOGLE, INC.: *AutoService*. <https://github.com/google/auto/tree/master/service>, April 2014. Accessed: 25.04.2018.
- [25] HANNES DORFMANN: *Annotation Processing 101*. <http://hannedorfmann.com/annotation-processing/annotationprocessing101>, January 2015. Accessed: 25.04.2018.
- [26] IDC: *IDC: Smartphone OS Market Share*. <https://www.idc.com/promo/smartphone-market-share/os>, 2017. Accessed: 18.05.2018.
- [27] IRONIN: *Explaining GraphQL and How it Differs from REST and SOAP APIs*. <https://www.ironin.it/blog/explaining-graphql-and-how-it-differs-from-rest-and-soap-apis.html>, October 2017. Accessed: 14.05.2018.

- [28] JETBRAINS: *Comparison to Java - Kotlin Programming Language*. <https://kotlinlang.org/docs/reference/comparison-to-java.html>, 2016. Accessed: 28.04.2018.
- [29] JETBRAINS: *Extensions - Kotlin programming Language*. <https://kotlinlang.org/docs/reference/extensions.html>, April 2016. Accessed: 25.04.2018.
- [30] JETBRAINS: *Higher-Order Functions and Lambdas - Kotlin programming Language*. <https://kotlinlang.org/docs/reference/lambdas.html>, April 2016. Accessed: 25.04.2018.
- [31] JETBRAINS: *Using Gradle - Kotlin Programming Language*. <https://kotlinlang.org/docs/reference/using-gradle.html>, April 2016. Accessed: 25.04.2018.
- [32] JETBRAINS: *Annotation Processing with Kotlin - Kotlin Programming Language*. <https://kotlinlang.org/docs/reference/kapt.html>, May 2017. Accessed: 25.04.2018.
- [33] JETBRAINS: *walkTopDown - Kotlin Programming Language*. <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/java.io.-file/walk-top-down.html>, November 2017. Accessed: 17.05.2018.
- [34] LEE BYRON: *GraphQL: A data query language*. <https://code.facebook.com/posts/1691455094417024>, September 2015. Accessed: 21.04.2018.
- [35] LUCCHI ROBERTO, MILLOT MICHEL, ELFERS CHRISTIAN: *Resource Oriented Architecture and REST*. http://inspire.ec.europa.eu/reports/ImplementingRules/network/Resource_orientated_architecture_and_REST.pdf, June 2008. Accessed: 14.05.2018.
- [36] MATT TEA: *A Massive Guide to Building a RESTful API for Your Mobile App*. <https://savvyapps.com/blog/how-to-build-restful-api-mobile-app>, July 2017. Accessed: 14.05.2018.
- [37] METEOR DEVELOPMENT GROUP: *Apollo GraphQL | Learn about the Apollo platform: Client, Engine, GraphQL servers, GraphQL support, and more..* <https://www.apollographql.com/>, 2014. Accessed: 21.04.2018.
- [38] METEOR DEVELOPMENT GROUP: *Downloading a schema | Apollo iOS Guide*. <https://www.apollographql.com/docs/ios/downloading-schema.html>, October 2016. Accessed: 24.04.2018.
- [39] METEOR DEVELOPMENT GROUP AND CONTRIBUTORS: *Apollo GraphQL Client for Android*. <https://github.com/apollographql>, 2014. Accessed: 24.04.2018.
- [40] METEOR DEVELOPMENT GROUP AND CONTRIBUTORS: *Apollo GraphQL Client for Android*. <https://github.com/apollographql/apollo-android>, September 2016. Accessed: 21.04.2018.

- [41] MIQUEL BELTRAN: *Hello World of Annotation Processing in Kotlin*. <https://proandroiddev.com/hello-world-of-annotation-processing-in-kotlin-3ec0290c1fdd>, September 2017. Accessed: 17.05.2018.
- [42] RXJAVA CONTRIBUTORS: *ReactiveX/RxJava: RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM..* <https://github.com/ReactiveX/RxJava>, 2016. Accessed: 17.05.2018.
- [43] SCOTT BOLINGER: *Is a mobile app better than a mobile website?*. <https://appresser.com/app-better-than-website/>, February 2015. Accessed: 14.05.2018.
- [44] SQUARE INC.: *OkHttp*. <http://square.github.io/okhttp/>, September 2011. Accessed: 26.04.2018.
- [45] SQUARE, INC.: *square/okhttp: An HTTP+HTTP/2 client for Android and Java applications..* <https://github.com/square/okhttp>, July 2012. Accessed: 26.04.2018.
- [46] SQUARE, INC.: *Retrofit*. <https://square.github.io/retrofit/>, 2013. Accessed: 29.04.2018.
- [47] SQUARE, INC.: *retrofit/retrofit-converters at master · square/retrofit*. <https://github.com/square/retrofit/tree/master/retrofit-converters>, 2013. Accessed: 29.04.2018.
- [48] SQUARE, INC.: *square/retrofit: Type-safe HTTP client for Android and Java by Square, Inc..* <https://github.com/square/retrofit>, 2013. Accessed: 26.04.2018.
- [49] SQUARE, INC.: *Interceptors · square/okhttp Wiki*. <https://github.com/square/okhttp/wiki/Interceptors>, December 2014. Accessed: 26.04.2018.
- [50] SQUARE, INC.: *Recipes · square/okhttp Wiki*. <https://github.com/square/okhttp/wiki/Recipes>, June 2014. Accessed: 26.04.2018.
- [51] SQUARE, INC.: *HTTPS · square/okhttp Wiki*. <https://github.com/square/okhttp/wiki/HTTPS>, March 2015. Accessed: 26.04.2018.
- [52] SQUARE, INC.: *retrofit/retrofit-adapters at master · square/retrofit*. <https://github.com/square/retrofit/tree/master/retrofit-adapters>, 2015. Accessed: 29.04.2018.
- [53] SQUARE, INC.: *Converter.Factory (Retrofit 2.3.0 API)*. <http://square.github.io/retrofit/2.x/retrofit/retrofit2/Converter.Factory.html>, 2017. Accessed: 29.04.2018.
- [54] SQUARE, INC. AND CONTRIBUTORS: *A Kotlin API for generating .kt source files..* <https://github.com/square/kotlinpoet>, December 2012. Accessed: 25.04.2018.
- [55] TODD FREDRICH: *RESTful Service Best Practices*. https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf, August 2013. Accessed: 14.05.2018.

List of Listings

Listing 1	JSON object of a sample Person object	2
Listing 2	Minified JSON object of a sample Person object	3
Listing 3	Example GraphQL Query	3
Listing 4	Example GraphQL Response	4
Listing 5	Example of a custom scalar type definition	5
Listing 6	Example of an object type definition	5
Listing 7	Example of an enum type definition	6
Listing 8	Example of lists and non-nullable types	6
Listing 9	Example of a Query and Mutation type definition	7
Listing 10	Example of an Introspection Query	8
Listing 11	Example of an Introspection Query Response	8
Listing 12	ExampleFeedQuery.graphql	11
Listing 13	Sample Apollo Client usage	12
Listing 14	Sample Kotlin lambda receiver	15
Listing 15	Usage of the lambda receiver	15
Listing 16	Example of a lambda parameter as single parameter and omitted brackets	15
Listing 17	Sample Kotlin Poet Code Generation	16
Listing 18	Sample Annotation Processor construct using Kotlin and the AutoService annotation	18
Listing 19	Sample HTTP request using OkHttp	18
Listing 20	Creating a schema using the Kotlin DSL	20
Listing 21	Create a Query using a Kotlin DSL provided by generated classes	22
Listing 22	Generated query string from a query created using a Kotlin DSL provided by generated classes (added line indents for better readability)	22
Listing 23	Example of retrieving a list of all files with the (case-insensitive) extension graphql	23
Listing 24	Example usage of the GraphQLConverterFactory by building a retrofit instance	24
Listing 25	Usage of the Converter in a retrofit interface	24

List of Abbreviations

DSL	Domain Specific Language
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
REST	Representational State Transfer
npm	Node Package Manager
APT	Java Annotation Processor
KAPT	Kotlin Annotation Processor
API	Application Programming Interface
IDL	Interface Definition Language
JSON	JavaScript Object Notation
URL	Uniform Resource Locator