Vladimir Maskov

Implementing REST Client for Android

Helsinki Metropolia University of Applied Sciences

Degree Bachelor of Engineering

Degree Programme Information Technology

Thesis

8 April 2015

Author(s) Title Number of Pages Date	Vladimir Maskov Implementing REST Client for Android 48 pages + 3 appendices 8 April 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Development
Instructor	Peeter Kitsnik

The aim of this final year project is to explore various ways to build an Android REST client by evaluating existing solutions and approaches. The project introduces several popular REST client libraries and persistence options, which allow to simplify an interaction with RESTful Web services and enable important features, such as caching, response parsing and result filtering. Object relational mapping and networking solutions are covered in detail.

The practical result of the study is illustrated in a start-up application developed since May 2014 to demonstrate the whole development process from back-end solution selection until the product is ready for the market. The materials are presented in the form of analysis, extensive code listings and figures. Several approaches and products have been compared to provide comprehensive insight according to the discovered facts.

The main components and resources utilized by the application are SQLite, BaasBox Android SDK and custom Content Provider working in a pair with SyncAdapter. SQLite and BaasBox were replaced with Realm and Retrofit in the later part of the work due to certain constraints and performance issues. Modularity of an application is suggested as a key element essential for successful development. Information gathered during development is analysed and the most efficient solutions are planned to be applied in future projects.

Keywords	REST, HTTP, Android, SQLite, JSON, XML

Contents

1 Introduction

History of Representational State Transfer (REST) begins from the moment when it was initially described by Roy Fielding in his paper "Architectural Styles and the Design of Network-based Software Architectures" [1] in 2000 as a concept of web architecture for distributed computing. Nowadays it has become a popular standard used by market leaders (Google, Facebook) and has mostly substituted the Simple Object Access protocol (SOAP) and the Web Services Description Language (WSDL) due to its simpler and lighter architecture. REST frameworks and Application Programming Interfaces (APIs) continuously develop. REST is a high-level concept that could be adopted by the use of a wide technology stack to handle various types of data. Due to the abstract nature of the REST concept, there are no exact specifications about methods and their certain behavior. Standard implementation of the REST concept is HyperText Transfer Protocol (HTTP-based, it relies on Web architecture and utilizes HTTP verbs for a uniform interface). Restful API provides a flexible way to represent a service's resources for different applications in a standard data format, which simplifies integration for complex systems where data is mixed or combined. The RESTful Web service architecture is required to follow four basic design principles: explicit use of HTTP methods, statelessness (application state is never stored on the server), URIs are represented in a format, which is similar to the directory path structure on a PC, data is transferred in eXtensible Markup Language (XML), JavaScript Object Notation (JSON), or in both formats. [2]

The content of the thesis primarily focuses on a client side of the REST services, their adoption on an Android mobile platform and real-life implementations in production. However, the REST client topic would not be covered completely without a basic knowledge of essential design principles and implementation of methods on the server side. Several REST patterns presented by Google are covered in chapter 3 conjointly with a list of libraries, necessary for optimal implementation of the client on an Android platform. Persistence is the main topic of chapter 4, which provides important information about a developer choice of various storage solutions. Traditional SQLite database, its ORM add-ons and modern Realm.io solution (which is just gaining its popularity in 2015) are compared and discussed. Practical work is mainly based on a startup project work carried out since May 2014 until January 2015. The project passed through several evolution stages while knowledge and experience were evolving during the development and preparation of the thesis. All stages are covered in chapter 5, and extensive source code listings are placed in appendices 3 and 4.

2 Restful API Architecture

2.1 HTTP Methods

A RESTful Web service should always use HTTP methods explicitly according to RFC 7231. For example, HTTP GET, is supposed to be used by a REST client to retrieve data from a web service or to execute a query to fetch a result set which consists of the requested resources. [3, 23] The fundamental REST design principle is exact mapping between create, read, update, and delete (CRUD) operations and REST HTTP methods. Mapping should follow certain rules:

- POST creates a resource on the server [3, 25].
- GET retrieves an appropriate resource [3, 23].
- PUT should be used to update the resource or change the state [3, 25].
- DELETE is used only to remove the resource [3, 28].

However in the case of incorrect architecture design Web APIs use HTTP methods for unintended and unexpected (for a client-side developer) purposes: transactions can be triggered on the server and records modified or deleted from remote database.

2.1.1 HTTP GET

The HTTP GET method invocation may look like the one shown below.

```
GET /addproduct?title=mangos&amount=35 HTTP/1.1
```

Listing 1. An example of an incorrect HTTP request

HTTP GET method represented in listing 1 is not used correctly and does not follow the original REST approach. The HTTP/1.1-compliant GET request by design returns resource representation as a response and is not designed to serve as a trigger to add a record to a remote database. The requests are used only to read the data and not change it. If the data is modified, a request is considered unsafe. If it could be executed without a risk of data modification or removal, calling it once has the same effect as calling it multiple times, or none at all. Furthermore, GET is idempotent, which means that making multiple requests using the same URI must produce the same result as just a single request.

```
GET http://products-example.com/products/vegetables
GET http://products-example.com/products/fruits
GET http://products-example.com/products/vegeta
bles/cucumbers
```

Listing 2. Examples of an HTTP GET request

If the request is executed successfully (URI is correct), GET will return a representation in XML or JSON and the HTTP response code of 200 (OK) [3, 57]. In the case of a failed request, 404 (NOT FOUND) [3, 65] or 400 (BAD REQUEST) [3, 64] errors are returned. Listing 2 illustrates examples of valid GET requests.

2.1.2 HTTP POST Request

The HTTP POST method is used mostly to create new resources. When a new entity is being created, the service associates it with a parent and assigns an ID (new resource URI). If the resource is created successfully, the web server returns the HTTP status code 201 and a link to the newly-created entity in a Location header [3, 53-54]. Listing 3 illustrates how new resources are created via HTTP POST request. POST is not considered safe nor idempotent and is supposed to be used for non-idempotent HTTP requests [3, 74]. However when POST requests containing an equal URI and the same entities are repeated, the server creates resources which contain the same data under the common parent. [3, 50-51].

```
POST http://products-example.com/products/vegetables
POST http://products-example.com/products/fruits
```

Listing 3. Examples of valid HTTP POST request

If HTTP GET is unsafe, Web caching tools and search engines (for example, Google) can execute data changes accidentally. To eliminate that problem parameter-value pairs in the request URI could be bundled into the XML entity. The created XML representation may be delivered in the body of an HTTP POST request, where the URI serves as a parent of an entity and the xml entity is a payload. Listing 4 demonstrates how data can be embedded into the request body. If the chosen format is xml, the content-type field should be set to “application/xml”; by default it is “plain/text”. [2]

```
POST /products/vegetables HTTP/1.1
Host: products-example.com
Content-Type: application/xml
<?xml version="1.0"?>
<product>
  <title>Cucumbers</title>
  <amount>15</amount>
</product>
```

Listing 4. XML payload in HTTP POST request body

The client defines the relationship by URI in the HTTP POST request. When the request is received on the server side, new resource is added to the parent “vegetables”. This relationship between the parent and new entity specified in HTTP POST is similar to the tree structure of the file system.

2.1.3 HTTP PUT

HTTP PUT is used mostly for updating certain resources, which are available via an existing URI. The request body includes an updated version of an entity already stored on the server. If the request was successful, the 200 OK result code should be returned. If the response does not contain an embedded body then 204 code is used instead of 200 code [3, 25-26].

HTTP PUT will create a new entity on the server in case the supplied id of the resource does not exist, yet. This behavior does not seem obvious to every developer and is recommended to be avoided in order to eliminate problems and confusion. When it is needed to create new resource, the HTTP POST request should be used instead of HTTP PUT. There is also an alternative way: the resource id can be determined in the entity’s body and the URI should skip the resource id part. [2]

When HTTP PUT is used to create new resource on the server, HTTP code 201 is returned if the request did not fail. The body is usually not included in the response (the client keeps the resource id in memory) [3, 26].

```
PUT http://products-test.com/vegetables/egg_plants
PUT http://products-test.com/fruits/oranges
```

Listing 5. Examples of valid HTTP PUT request

However HTTP PUT is not considered safe [3, 74] (because the state of data on the server is modified when the request is processed), but if one of the statements presented in listing 5 is repeated multiple times, the entity stored on the server does not change and remains in the same state as after the initial request. This makes HTTP PUT an idempotent operation, but in a certain case the request can become not idempotent. That happens if the value is incremented and delivered to the server using HTTP PUT. Using the HTTP PUT method only for not idempotent operations helps to avoid mistakes and inconsistency. This is considered the recommended approach according to documentation. [2]

2.1.4 HTTP DELETE

The HTTP DELETE method operation is quite straightforward: it allows to remove data from the server using a resource identifier. If the resource is removed successfully, HTTP DELETE returns HTTP code 200 in response, if not - code 204 is returned [3, 28-29]. Sometimes the body may be attached and include removed entity. For example, when “potatoes” are deleted from the server in listing 6, the response may contain details about the removed item – weight, price, quality and other information.

```
DELETE http://products-test.com/vegetables/potatoes
DELETE http://products-test.com/fruits/lemons
```

Listing 6. Valid HTTP DELETE requests

According to specifications, the HTTP DELETE method is considered idempotent. On the first request, the resource is removed and if that operation is repeated multiple times, the result is the same. [2] The HTTP DELETE method should be kept idempotent and used only to remove a certain entity on the server-side, however if data on the server is already removed, executing one of the statements again from listing 6 will produce 404 responses [3, 58], because the resource is not reachable anymore.

2.2 REST URI Format

The URI scheme of REST API [4, 16] is especially important for a client-side representation of resources addressed on the server. The URI should be designed in such a manner that it is relatively simple and intuitive to encode paths to the Web API resources without or with minimal amount of extra information (documentation, references). The URI scheme endpoints should be clear to understand and highly predictable.

The most straightforward approach is to accommodate directory structure-like URIs: the address is built in strict hierarchy, resources have a single common root (base URI) and they include multiple branches (paths). The URI could be imagined as not just a simple string, which is delimited, as a tree with branches which represent resource main categories. [2] For example, all books in a web shop can be represented in the form of a URI structure as shown in listing 7.

```
http://products-test.com/books/genres/{genre}
```

Listing 7. URI structure used to present books by genres

The node “/genres” is attached to the root “/books”. Under the “/books” branch there are several genres, such as sci-fi, romance or drama and each genre points to a set of books. Using this clear intuitive structure it is easy to guess a certain book URI. Resources can be organized hierarchically based on predefined rules, for example by title as done in listing 8.

```
http://products-test.com/articles/2014/11/18/{title}
```

Listing 8. Hierarchical URI structure

This URI format looks intuitive and makes a perfect directory-like structure. The first part of the path attached to “/articles” is a four-digit year, the second is a two-digit month and the third - a two-digit day. The URI in listing 9 is well-formed and keeps a human-readable format entirely and at the same time can be encoded by the server using predefined rules.

```
http://products-test.com/arti-
cles/{year}/{day}/{month}/{title}
```

Listing 9. Complex URI in human-readable format

In a properly designed REST API URIs remain static and can be bookmarked. Even if implementation changes on the server-side, client endpoints and relations between resource nodes do not change.

2.3 Statelessness

Two state types exist in a service which is built on the REST architecture. One is a resource state, which stores resource information and application state – the information about the resource path [5, 90]. The RESTful service holds only resource state and sends it to the REST client as entities. Clients keep an application state, which can be used for all CRUD operations. The application state can be transferred to the server via HTTP POST, PUT and DELETE requests. On the server side it is transformed into a form of resource state. The stateless nature of the REST service introduces server-side limitations: the application state is saved only on the client side. The stateless REST service may consider the application state, but only as a part of the HTTP request, which may include a session id or account credentials (for example, user name and password hash), which should be sent with each request. [5, 217]

The stateless REST service has better performance, simpler design and scalability because it does not hold and manage the application state. The REST server only produces responses to client requests and enables the interface to help the client manage the application state. [5, 86] For instance, the client should include a certain title of the book to fetch instead of requesting the next one. The client’s application state is managed by sending representations; the server’s resource state is changed by submitting a representation. The client handles resource state by sending a representation via POST or PUT. [5, 218] Collaboration between the client and the server is an important feature which makes a RESTful service stateless. It also helps to save bandwidth between those components by minimizing the amount of data transmitted.

Each request sent to the server includes all the required information (including application state) to process data and never considers requests accomplished before. All data stored on the server should be represented in the form of a resource accessible via URI. States stored on the server are considered resources and exposed via URI: "The client should not have to coax the server into a certain state to make it receptive to a certain request” [5, 87]. Some web services do not follow a stateless approach and restrict the order of client requests. If this happens, the user is not able to navigate between previous, current and next pages (states). It could be confusing when the user presses the backbutton in the browser or repeats the request.

Resource representation carries the state of a resource, multiple attributes including exact time when the resource was requested by the client and acts like a snapshot. As a good example, a parallel with a database record or data model could be provided, where resource representation is a snapshot of various attributes requested by a REST client.

2.4 Data Transfer Format

In order to transfer data from the RESTful Web service to a REST client, the data format should be chosen to determine the request and the response HTTP body payload. It is highly important to keep relationships between resources while transferring data objects. [2] In the notification service author names, heading and message body can be provided all together in a response to a certain requested resource as illustrated in listing 10.

```
<?xml version="1.0"?>
<message>
<to>John </to>
<from>Jane</from>
<heading>Reminder</heading>
<body>Hello, my friend</body>
</message>
```

Listing 10. XML representation of a message

For RESTful web application is recommended to use the HTTP Accept header [3, 37] and specify the MIME type to request an appropriate content type. Table 1 includes specific MIME types used by RESTful Web services.

Table 1. Common MIME types used by RESTful services [2]

MIME-Type	Content-Type
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

The MIME types allow to make the service really cross-platform and target maximum number of various platforms and devices. However, in certain cases it is enough to provide only one of the formats (when the number of platforms is limited and the clients require a specific content type).

3 Patterns and Solutions for REST client implementation

3.1 Implementation Patterns

REST client implementation on Android has a few key features. Various approaches were extensively described by Virgil Dobjanschi on Google I/O 2010 developer conference [6]. Google developed three correct ways of REST client implementation.

Pattern A. Use Service API: Activity <-> Service <-> Content Provider [7, 366]. In this case, Activity works with Android Service API (as illustrated in Figure 1). Whenever the REST request is ready to be sent Activity creates Service then Service asynchronously passes requests to REST-server and saves results in the Content Provider (SQLite). Activity automatically receives notification about the request completion status and reads data from the Content Provider (SQLite).

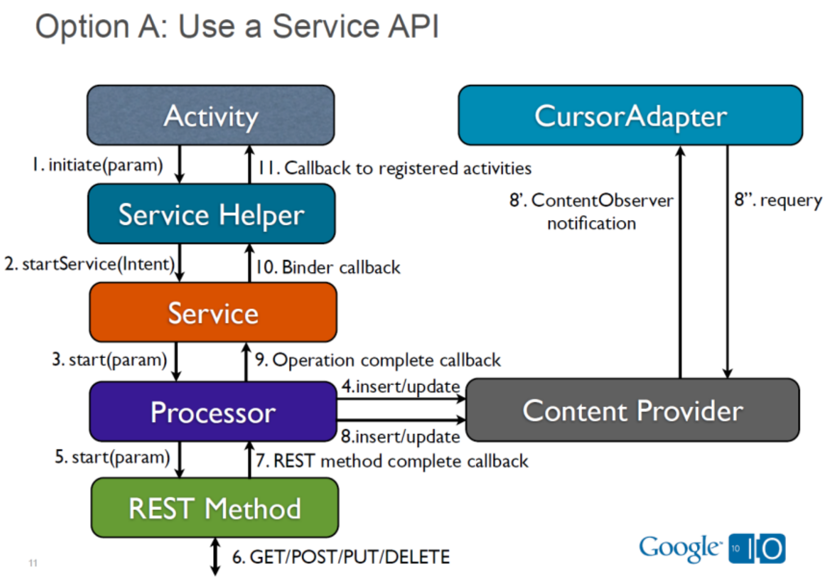


Figure 1. Pattern A presented at Google IO 2010 [6, 11]

Pattern B. Use Content Provider API: Activity <-> Content Provider <-> Service [7, 367]. In this case, Activity works with Content Provider API: Activity <-> Content Provider <-> Service, which acts as a facade for the Service (as illustrated in Figure 2). This approach is based on common features of Content Provider API [8] and REST API: HTTP GET REST request is equivalent to SELECT query to database, HTTP POST REST ~ INSERT, HTTP PUT REST ~ UPDATE, DELETE REST ~ DELETE. Activity loads results from SQLite database (as done in pattern A).

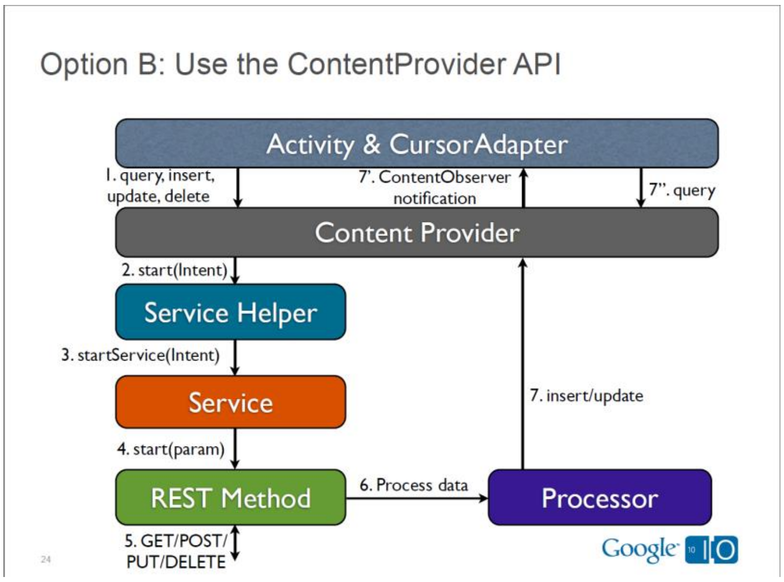


Figure 2. Pattern B presented at Google IO 2010 [6, 24]

Pattern C. Use Content Provider API and Sync Adapter: Activity <-> Content Provider <-> Sync Adapter [7, 367]. Pattern C is a modification of Pattern B where SyncAdapter is used instead of Service (as shown in Figure 3). Activity queries Content Provider to get records from the database and local data itself is synced with a remote server using the onPerformSync method of the Sync Adapter class [9].

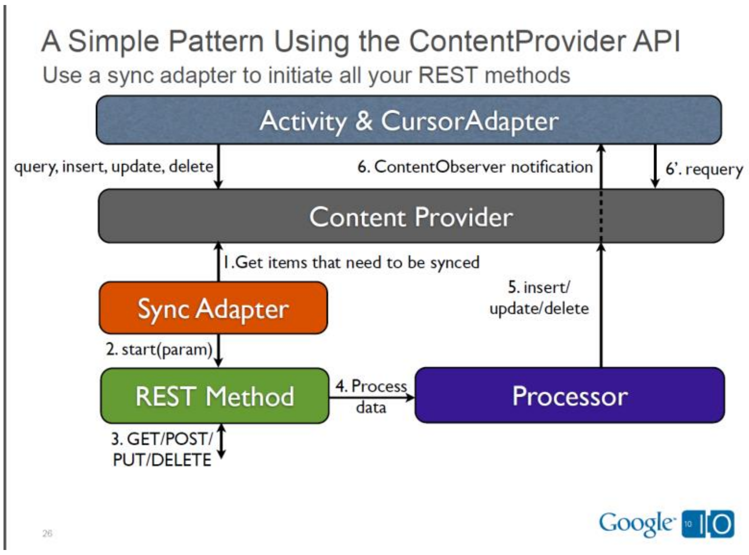

A Simple Pattern Using the ContentProvider API
Use a sync adapter to initiate all your REST methods
Activity & CursorAdapter
query, insert, update, delete
6. ContentObserver notification
6'. requery
Content Provider
1.Get items that need to be synced
Sync Adapter
5. insert/update/delete
2. start(param)
REST Method
4. Process data
Processor
3. GET/POST/PUT/DELETE
26
Google IO

Figure 3. Pattern C presented at Google IO 2010 [6, 26]

Sync Adapter is executed indirectly from Sync Manager and there are several options to run the Sync Adapter according to documentation:

1. “Run the Sync Adapter When Server Data Changes
2. Run the Sync Adapter When Content Provider Data Changes

3. Run the Sync Adapter After a Network Message
4. Run the Sync Adapter Periodically
5. Run the Sync Adapter On Demand” [9]

It is important to notice that according to Android documentation the Sync Adapter should not be run as the direct result of a user action, because in this way the application does not get full benefit of the Sync Adapter approach. Google emphasizes that a developer should avoid providing a refresh button in user interface when using Sync Adapter.

The easiest approach which could be implemented (and used by beginner level programmers), is totally incorrect: a separate thread is executed from Activity, the data is sent to the REST server via a request, and the results are saved in memory (for example, as the Array List structure without a database and any kind of persistence). Due to its unreliable nature this approach should not be used for production. Figure 4 illustrates incorrect implementation, which does not handle persistence.

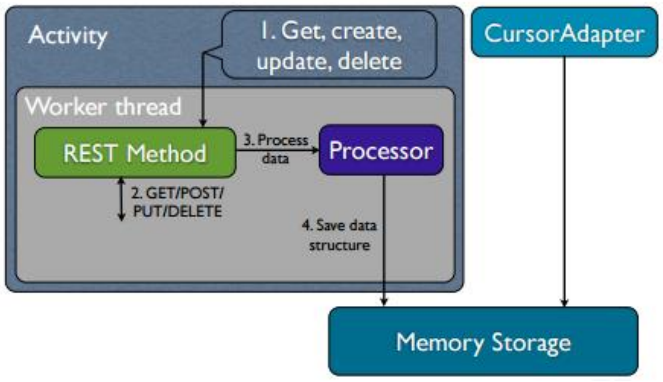


Figure 4. Incorrect REST client implementation [6, 6]

Most of the problems are related to Activity Lifecycle - an order of states, which activity passes in its lifecycle, cannot be predicted due to user interactions. Activity can be paused or even destroyed at any time by Android OS. It means that it is impossible to reliably execute long-running operations in the Activity code, because it can be destroyed before the results are received and the operation finished. There could be two consequences of such behavior: results are lost and data is desynchronized between the server and the client. Therefore, long-running operations should be executed in the Service. The service has a higher priority than the Activity and memory management will not stop for most cases. The Service and Activity are two different components of the Android framework and the data between them should be passed using serialization. As described by Virgil Dobjanschi in Google I/O 2010, there is a parallel with marshalling – using marshalling it becomes possible to pass data, but the amount of data should be minimized. An optimal approach is to store the data in the database and then read it back in the Activity.

In correct implementation of the REST client, a response received from the REST server is persisted and never passed directly to Activity. Activity is notified that the data is stored in the database and can be loaded (this can be achieved by using the Content Provider with the Content Observer). While INSERT, DELETE, UPDATE operations are being executed, the data in SQLite is updated twice: the first time before a REST request is sent, the second time when the HTTP response is received. [8] The first operation sets status flags, signaling about the type and status of the executed operation. The REST methods should always be executed in a separate thread and the Apache Http Client is a preferred due to various bugs in HttpUrlConnection class implementation on different Android SDK versions [10].

3.2 Important Details and Optimizations

Several ways to optimize a REST client operation were presented in Google IO 2010 [6]:

- Gzip (GNU Zip) [4, 38] compression brings benefits if used in REST-client implementation. The Gzip library is included in native Android SDK and it helps to minimize traffic, accelerate data reception, save battery power.
- When data is stored in the SQLite database, transactions would increase the overall speed. If the application is required to download more than 10 images, it is better to start only 1-3 parallel downloads and queue for others. It allows the first images to appear faster and leaves a certain percentage of bandwidth to other applications.
- The Activity registers a binder callback (Result Receiver) to get a response from the Service. This callback should be removed when onPause is called in the Activity lifecycle, otherwise the Application Not Responding dialog may appear.
- Long-running operations should always be executed from the Service. Service should be always stopped when requested operations finished.
- Database should not grow infinitely, old records have to be removed (for example, by timestamp).
- Data should be paginated (if REST API supports that).

- If time is not a critical parameter, data should be synchronized using the Sync Adapter framework.

There are also certain problems which should be resolved when implementing a REST client:

- Manage Service: start/stop.
- Pass data from Service to Activity.
- Cache results in SQLite database.
- Save status of data before and after REST-request.
- Record information about current REST-operations in SQLite database.
- Parse received response.
- Build REST-request based on encoded URI and set of parameters.
- Execute REST-requests to server.
- Database cleaning to save space from stale data.
- In case of fail, unsuccessful REST-request should be repeated using exponential back-off timer.
- Possibility to execute REST request via Sync Adapter.

The presented patterns can be implemented manually or using external libraries to build the REST client. The simplest solution is to use one of the external libraries described in following sections.

3.3 REST Client Libraries

3.3.1 Robospice

RoboSpice is a modular Android library that simplifies the process of handling asynchronous requests [11]. RESTful Web services are supported out-of-the box. Request cancellation, request prioritization and request aggregation are supported. The library leverages caching and has an external cache option available (it is required to implement an abstract class CacheManager). There are several options for cache format: JSON (Jackson/Jackson2/Gson supported), XML, plain text or binary data. When a request is being executed, a cache option can be set on/off, cache time (the time when the results are valid). Among important library extensions there is an ORMLite module used to write and read POJO to and from SQLite. Data identification is based on class implementations.

The Robospice library is strongly typed; therefore POJOs are used as request parameters and POJOs are received back as request results. The library is well tested (the repository includes more than 200 tests) and has extensive documentation; thereby stability and efficiency are guaranteed. [11]

Extra information gathered from library classes published in the repository includes the following:

- There is no built-in support for pre- and post- operations of the REST methods, but it is possible to create implementation using a class derived from SpiceRequest.
- A network connection is customized and configured via a class derived from SpiceRequest (loadDataFromNetwork method). The library includes implementation of SpiceRequest by default based on java.net.URL for plain text data and HttpURLConnection [10] for binary data.
- A retry algorithm is configured via RetryPolicy. DefaultRetryPolicy used by default implements an exponential back-off algorithm.
- The data is passed from Activity to Service using RequestListener. The result type is customized using a generic parameter. A listener is passed as a parameter into spiceManager.execute(), the results are received in a parsed representation.

3.3.2 Datadroid

According to a repository hosted on GitHub [12], the Datadroid purpose is to ease the data management in an Android application. Brief documentation includes several steps to set up a library project (no gradle or maven support). The repository includes a sample, but to integrate the library properly, a fair amount of research is required (classes in the repository, external resources), which means that Datadroid cannot be used by a beginner developer. Unfortunately, the library is not in active development anymore (the last commitment on 10 March 2014), but it can still be used and adopted by an Android developer with a high level of expertise.

Extra information gathered from library classes published in the repository includes following:

- Cache is embedded in the RequestManager, data is stored as an LruCache object. For every type of the request caching is set separately. It is not possible to use database instead of LRU cache (it cannot be disabled).
- There is no built-in support for pre- and post- operations of the REST methods, but it is possible to create an implementation using a class derived from Operation.
- Network resources accessed via OkHttp from Square or HttpURLConnection. The first options are used through java.lang.reflect and if an exception is raised, default HttpURLConnection will be used.
- A retry algorithm is not available. In the case of an error during the request execution, an exception is raised with detailed information about the problem.
- Data is passed from Activity to Service using RequestListener. The results are loaded into Bundle. A listener is passed as a parameter into the execute method. The results are received as a bundle.

3.3.3 RESTdroid

RESTDroid [13] provides similar features as previous libraries and follows a modular approach (like RoboSpice). The library includes a necessary functionality to handle requests, but additional features are also available via external modules, which can be found in the RESTDroid repository. For example, the ORMlite-Jackson module handles data persistence and mapping/parsing using the JSON Jackson format and the ORMLite framework (described in chapter 4). Extensive documentation and beginner guides are published on the site of the developer. [13]

Extra information gathered from library classes published in repository includes the following:

- Cache is embedded (CacheManager), cache results are stored in separate files. Validity of data in the cache is determined by file creation time. It is possible to disable cache reading, but not cache writing. Storage method can be modified by changing the implementation of the object inherited from PersistableFactory.
- Logic of pre- and post- requests can be modified using the derived class Processor which has overridden the methods preRequestProcess and preGetRequest.

- The Apache HTTP client is used to access network resources (for Android 2.3 and higher HttpURLConnection is the preferred implementation). Automatic retry is done after a predefined time interval (by default after 1 minute).

3.3.4 Retrofit

Retrofit is a modern type-safe REST client library for Android and Java created by Square Inc (2013-2015). It provides a convenient way for authenticating and interacting with various APIs and allows sending network requests with OkHttp or HttpUrlConnection. The library fetches JSON or XML data from the RESTful web service and once the response is received, it will be parsed as a Plain Old Java Object (POJO), which should be specified for the object in the response. Custom JSON parsers and the GSON utility library are supported for deserialization and auto parsing. Retrofit works with REST API using Java interface implementation, which could be generated with a help of RestAdapter. Implementation in this case acts like a local instance of the service and every call corresponds to the HTTP request. [14]

The library uses annotations extensively to specify how each request will be handled:

- For automatic URL parameter replacement and query string support.
- To support form-encoded and multipart data in request body.
- To enable file uploads.
- To set custom headers.
- To set a remote method relative path.

Retrofit annotations and integration of the library into the project are covered in more detail in chapter 5 with a focus on practical matters.

3.3.5 Comparison of REST Client Libraries

Table 2 illustrates common features of REST-client libraries discussed in sections 3.3.1 – 3.3.4. This section includes a comparison and summary about the most popular solutions.

Table 2. Common features of REST-client libraries

Library	**RoboSpice** [11]	**Datadroid [12]**	**RESTDroid [13]**
Pattern type	A	A	A
Data cache	External	Embedded	Embedded
Identification of REST-request types	-	int	UUID
Service Helper layer	SpiceManager	RequestManager	WebService
Pre- and post- operations for REST methods	No built-in support	No built-in support	+
Embedded tools for parsing results	Method loadData-FromNetwork in SpiceRequest<T> supports parsing implementation	It is possible to create implementation of the parser (there is an example in sample application)	Parsing of results is supported via parseToObject in Processor
Automatic retry in case of failed request	Available	Not available	Available
Gzip support	Is implemented via class customization SpiceRequest	Is embedded, configured via setGzipEnabled in NetworkConnection class	Not implemented (HttpRequestHandler class).
REST-request builder	All standard implementations of SpiceRequest<T> accepts encoded URL as a parameter (but it should be encoded before by external method)	Request class allows to set parameters of the request. NetworkConnectionImpl implements URL encoding for the REST-request.	RESTRequest accepts encoded URL as a parameter (but it should be encoded before by external method)
Notifications in UI thread about operation execution	Embedded (SpiceNotificationService).	No	No
Multithreading on REST-request send.	Size of thread pool is configured by overriding the method getThreadCount in SpiceManager and by default number of simultaneous threads is 3.	Size of thread pool is configured by overriding the method getMaximumNumberOfThreads in RequestService and by default only one thread is allowed.	Size of thread pool is configured by constant defined in WebService class, by default it is set as 10.
Embeded support of SyncAdapter	No (consequence of using pattern A)		
Minimal Android SDK version	8 (Froyo / 2.2)		
Sample applications	Samples are available in repository	Sample application DataDroidPoC is available	There is no sample included. Documentation and guide are available
Unit test availability	More than 160 test (as mentioned in description on github repository page).	No unit tests available	

Facts about REST client libraries can be summarized into following:

1) RoboSpice is a powerful, modular and well-documented library, which is updated frequently. It adopts Pattern A and has several useful features. Robospice is quite flexible library, which is suitable in the role of the fundamental element for a highly customizable solution.
2) RESTDroid implements pattern A and also has sufficient documentation.
3) Datadroid also uses pattern A, but does not have detailed documentation.
4) In RESTDroid and Datadroid customization is done based on generics (strict typing) and policies, common solutions are available. Both libraries include certain advantages and disadvantages and both follow the approach “plug and go”.
5) Retrofit. HTTP requests are described via annotations, the library supports synchronous and asynchronous REST method calls, and data can be transferred in JSON format or XML [14]. Retrofit is covered in more detail in chapter 5.

4 Object-Relational Mapping and Storage Solutions

4.1 Storage Solutions

4.1.1 SharedPreferences

SharedPreferences are considered the simplest and quickest storage solution for local data in Android framework. [15] It provides a straight way to store and retrieve defined key-value pairs associated with application. Each SharedPreferences file is managed by the framework and can be private, which allows data to be kept securely, or shared, in order to provide common data storage option. Unfortunately, because of its simplicity and minimalistic approach, SharedPreferences only can manage to save primitive data types (boolean, float, int, long, strings). This rule must be considered when choosing data types to save. [16, 8]. However, API 11+ supports sets of values to be saved in Set<String> format. SharedPreferences APIs can be paired with Preference APIs to create user interface for application settings screens based on reading and writing key value pairs.

Preferences data can be stored in a single file or in several separate files (quantity will depend on the number of activities):

- getSharedPreferences() — used if application needs several shared preference files identified by unique name. This method can be called from any Context in Android application.
- getPreferences() — used if only one shared preference file is needed per activity. This method does not require unique name as a parameter (preference is bind to Activity automatically). [15]

Listing 11 describes how to store and retrieve small pieces of data easily in a concise manner.

```
// Get instance of SharedPreferences
SharedPreferences pref = getSharedPrefer-
ences(“com.cloudnotify”, Context.MODE_PRIVATE);
```

```
// Retrieve SharedPreferences editor object
Editor editor = pref.edit();
editor.putString("user_id", "user@gmail.com");
editor.putString("token", “ar564645drtgd345”);

// This way set of values can be easily saved using
Shared Preferences framework
Set<String> userName = new HashSet<String>();
values.add("First Name");
values.add("Last Name");
editor.putStringSet("user_name", userName);

// Commit changes – this is important to save data
e.commit();

// Get values back (default values defined)
String stringValue = pref.getString("user_id", "er-
ror");
boolean booleanValue = pref.getString("token", “0”);
Set<String> userName = pref.getStringSet(userName, new
HashSet<String>());
```

Listing 11. SharedPreferences typical implementation

The first argument specifies which shared preference mapping instance should be retrieved (several shared preference files can co-exist per application). The second argument of the method sets an access level of the shared preference instance that is being retrieved (MODE_PRIVATE modifier means that data is kept securely, which guarantees that preference can be read and changed only by application which originally created it). When shared preferences object is instantiated, key value pairs can be retrieved using methods such as getString(), getInt(), getBoolean(), getLong(), etc. Each method takes two parameters such as **key** and **default value** in case data was not saved with this key before. Shared Preferences can be updated in a similar manner: Editor object is retrieved, values are set with any of put*() methods and then changes are stored with commit(). Deleting shared preference is also done in simple way. Remove() method is called and then changes are finalized with commit(). [15]

However, Shared Preferences class does not provide sophisticated schemes to store large pieces of data. There are several frequent use cases where Shared Preferences act as a perfect solution:

1. Verify if user enters the application for the first time.

Developers often would like to provide guidance and hints when user enters app for the first time. Sometimes it could also be necessary to remember user set rating for the application on Google Play. In this case dialog can appear, for example, when application is started for second time.

2. Last sync time.

Android client application requires synchronizing and caching data with backend regularly in order to keep information up-to-date. Last sync time can be saved to check if new sync process should execute.

3. Store user login credentials.

When user passes authorization steps necessary data should be saved to keep account information. If REST server needs access token to provide access to certain resource, token string should be saved in REST client via Shared Preferences to send a request.

4. Remembering an application's state.

Functionality of applications relies on application state saved on a client side and managed by user; this state should be kept in order to provide best user experience possible. Messenger applications such as WhatsApp and Viber have dedicated sections where user may choose the notification ringtone and manage push notification settings and other user preferences.

5. Save high score and current level in games.

Usually application also requires to save complex data structures, which makes Shared Preferences no longer the best choice. When complexity raises and various data types should be saved, it is better to switch to the mobile database solution and organize all data.

4.1.2 SQLite

SQLite is an open source self-contained, server-less database, which supports standard SQL syntax, prepared statements and transactions out-of-the-box. The library is designed to handle many kinds of system failures, such as low memory, disk errors, and power failures. No extra configurations are required for SQLite databases and SQLite is bundled with all API levels. Several data types are supported by default on Android:

INTEGER - corresponds to long, REAL – double in Java/Android, TEXT – simple String. If application needs to save any other data type, it should be converted into appropriate format, but there is no automatic validation and integer can be stored as a TEXT string. [17]

As a starting point to make use of SQLite database in Android application, subclass of SQLiteOpenHelper [18] must be created as illustrated in listing 12. Constructor includes a call to super() method which takes database name (String) and database version (int) values. Each database table should have column with "_id" (BaseColumns._ID constant) name, which is going to be used as a primary key in a standard way [16, 21]. If there is no such column it will be problematic to pair SQLite database with Cursors and Content Provider. All operations which require database access should be done asynchronously. Otherwise UI thread could be blocked (database file is stored in file system).

```
public class SQLiteHelper extends SQLiteOpenHelper {
private static final String DB_NAME = "sqlite.db";
// Version number should be incremented when updating
application
private static final int DB_VERSION = 1;
// Table name is related to data which should be
     saved, no spaces allowed
public static final String TABLE_NAME = "table_name";
// Fields should be created
public static final String ID = "_id";
public static final String USER_NAME = "name";
public static final String USER_PHONE = "phone";
// Constructor which implements super() method
SQLiteHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
// Override annotation is important here
@Override
public void onCreate(SQLiteDatabase db) {
db.execSQL("CREATE TABLE " + TABLE_NAME + " (" + ID +
" INTEGER PRIMARY KEY AUTOINCREMENT," + USER_NAME + "
VARCHAR(25)," + USER_PHONE + " VARCHAR(15));");
  }
```

```
 @Override
 public void onUpgrade(SQLiteDatabase db, int oldVer,
int newVer) {
// Remove table if it exists
db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
// Create new table via onCreate() method
onCreate(db);
}
}
```

Listing 12. Building SQLiteOpenHelper for SQLite database

It is important to override two methods for creating and updating the database:

1) onCreate() – creates database on first access attempt and defines structure.
2) onUpgrade() – triggered if database version increments in application. Method is necessary to update or drop existing schema and possibly rebuild it using onCreate() method. [18]

SQLite queries are executed using the query methods of SQLiteDatabase instance. Several query parameters allowed to be accepted: projection, selection, columns, sorting, name of the table, fields. Complex queries can be constructed using SQLiteQueryBuilder, which provides built-in methods for building queries, or native SQL syntax (manual way). Every SQLite query returns a Cursor, which points to all the rows, fetched by the query. Cursor enables navigation between result rows and allows to retrieve column data. In example provided in Appendix 1, SQLiteDatabase object is obtained via SQLiteHelper instance. Then ContentValues class instance is used as a wrapper for values to provide a convenient way for insert, update and delete operations in SQLite table. There is no need to assign "_id" column value, because it is incremented automatically due to AUTOINCREMENT option. [16, 21-24]

Use of SQLite without object relational mapping (ORM) in complex project can lead to cumbersome code, which is hard to support and maintain. ORM provides higher level of abstraction, allows to access a relational database in object-oriented manner and eliminates the need to implement conversions between objects and database. Android includes Content Provider class [8] (which was shortly described in chapter about REST client patterns), but it does not simplify database management and can even make development more time consuming, if no any code generators were used. However, Content Provider allows integration of useful Android components such as CursorLoaders and SyncAdapter.

4.1.3 Realm

Authors of Realm introduce their innovative product: “SQLite was revolutionary when it launched in 2000 but developing mobile apps in 2014 is obviously a very different beast than it was 14 years ago, and our notion of what a “phone” or “app” is has also changed drastically. We saw a clear opportunity to provide a fresh start for data on mobile with an easier API and an architecture that benefits from the last decade of innovation in databases. In short, we’re the first mobile-first database.” [19]

Compared to iOS Core Data or other ORMs built on top of SQLite, Realm utilizes simpler API, it is totally thread-safe and has better performance on queries and write transactions (comparison graph is available in the end of the chapter in section "Summary"). Realm itself is not built on top of SQLite and includes its own persistence engine built for optimized performance. Core is written on C++ from scratch and provides memory-efficient access to application data by using Realm objects (which consume less resources than native objects). There is also optional persistence layer available that can manage object retrieval and storage automatically. RealmObjects act like regular objects, but support queries with parameters, relationships (Many-to-One, Many-to-Many), search index, primary key, transactions, built-in thread-safety. Realm supports the several field types: boolean, short, ìnt, long, float, double, String, Date and byte[]. Integer types short, int and long are mapped to long. Relationships can be established by including RealmList<RealmObject> in RealmObject. [19]

Realm supports iOS & Android platforms. Realm files can be shared across platforms and adopt the same object models for Swift, Objective-C and Java. That simplifies process of porting an app from one platform to another, because business logic is similar, and reduces expensive development time.

Realm has important features:

1) Plain old Java object (POJOs) can be easily converted to RealmObjects using subclassing as shown in listing 13. Class should also include setters and getters (can be generated automatically in Android Studio).

```
public class Message extends RealmObject {
private String messageText;
private long timestamp;
}
```

Listing 13. RealmObject and its fields.

2) Persistence is achieved via efficient transactions illustrated in listing 14.

```
Realm realm = Realm.getInstance(this.getContext());
// Transactions give you easy thread-safety
realm.beginTransaction();
Message msg = realm.createObject(Message.class);
msg.setMessageText("This is my message");
msg.setTimestamp(2545611661L);
realm.commitTransaction();
```

Listing 14. Realm transaction

3) Thread Safety

Realm guarantees that all operations with database are thread safe and ACID compliant. ACID (Atomicity, Consistency, Isolation, Durability) means that database transactions are processed reliably. A single logical operation on the data is called a transaction. All operations between beginTransaction() and commitTransaction() are considered a single transaction. Commiting multiple operations in a bulk is more efficient than handling each operation separately. Transaction guarantees that objects are in consistent state and carry all necessary properties.

4) Object relations are handled as described in listing 15. AttachedData is a RealmObject which is bound with Message object in relation.

```
public class Message extends RealmObject {
private String messageText;
private RealmList<AttachedData> attachedData;
}
realm.beginTransaction();
Message message = realm.createObject(Message.class);
```

```
AttachedData attachedData = realm.createObject(At-
tachedData.class);
message.setMessageText("This is message text");
message.getAttachedData().add(attachedData);
realm.commitTransaction();
// Changes can be also discarded instead of committing
realm.cancelTransaction();
```

Listing 15. Creating relations between RealmObject's

5) Database can be queried using various filter parameter (multiple supported). All filter conditions are self-descriptive and can be used simultaneously (as shown in listing 16).

```
// Queries use Builder to build up query conditions
RealmResults<Message> query = realm.where(Message.class)
.greaterThan("timestamp", 26264642)
.contains("messageText","this is").findAll();
// Queries can be filtered further
RealmResults<Message> messages = query.where()
.contains ("messageText", "some text").findAll();
```

Listing 16. Realm filter parameters: simple solution to filter results

Realm.io is being developed actively and many features are going to be added to Android version of Realm library in future releases (migrations, encryption, fine-grained notifications – already available on iOS), but it already became a nearly perfect persistence solution. Realm supports direct object copying from response retrieved by Retrofit and helps to get rid of the boilerplate code necessary to store data on the REST client when communicating with REST service. There is no need to create helpers. The object can be copied using copyToRealm() static method.

4.2 Mobile ORM Libraries

4.2.1 OrmLite

OrmLite provides lightweight functions to implement persistence in Java applications (not only Android) and eliminates the need for manual table creation introducing efficient abstract Database Access Object (DAO) classes and annotations. OrmLite requires to create database helper class that will extend OrmLiteSqliteOpenHelper class (similar to SQLiteOpenHelper) and override OnCreate and onUpgrade methods. [20] Usually it is enough just to copy the source code from example tutorial application to implement Helper. Library extensively uses annotations: each class, which should be persisted, needs to have @DatabaseTable and @Database as presented in listing 17.

```
@DatabaseTable(tableName="message")
public class Book {
public Book() {}
@DatabaseField(generatedId=true, id=true)
private String id;
@DatabaseField(dataType=DataType.STRING)
private String text;
}
```

Listing 17. Use of annotations in OrmLite

It is easier to retrieve helper (by calling getHelper method) if one of the following classes is implemented: OrmLiteBaseActivity<Helper> OrmLiteBaseService<Helper>, OrmLiteBaseListActivity<Helper> or OrmLiteBaseTabActivity<Helper>.

```
// Retrieve the DAO for Message.class
// DAO will be created or returned from cache
public Dao getDao() throws SQLException {
if (bookDao == null) {
bookDao = getDao(Message.class);
}
return bookDao;
}
```

Listing 18. Obtaining DAO object in OrmLite

In the case the developer decides to create custom Activity or Service implementation, it is required to maintain helper state manually by calling OpenHelperManager.getH-elper in onCreate method and OpenHelperManager.releaseHelper in onDestroy. After helper instance has been retrieved, it is possible to get appropriate DAO (as illustrated in listing 18). Using DAO database queries can be executed.

4.2.2 GreenDAO

GreenDAO follows different approach than OrmLite. In order to use greenDAO in Android project, a second project, called the “generator” is required (pure Java, not Android). Generator task is to create necessary boilerplate code based on initial data specific to developed application. GreenDAO generator library (greenDAO-generator.jar) and the Freemarker library (freemarker.jar) have to be included in generator's classpath. [21] Process is quite straightforward: executable Java class is created, required entities are added to project schema and code generation can be started by running the project (scheme is illustrated in figure 5).

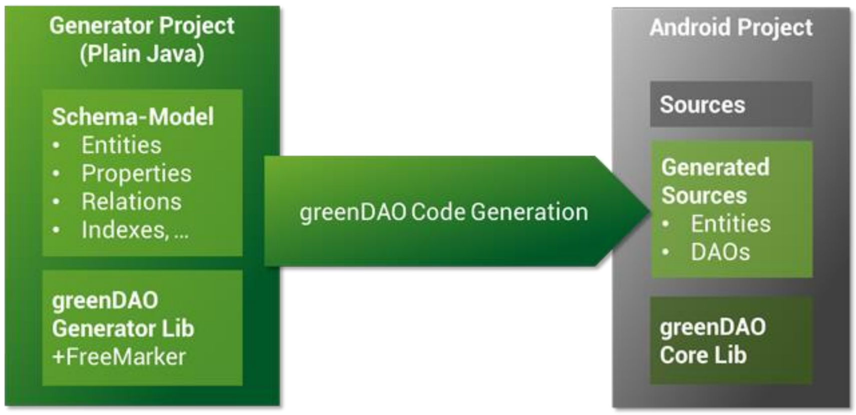


Figure 5. GreenDAO code generation scheme [21]

The code generation approach brings its own features:

- Improved performance (comparison is illustrated in following section 4.3).
- Tables are treated like entities (objects).
- Entities and boilerplate code required for SQLite are generated.
- Low memory consumption.
- No need to use annotations (although database schema should be generated).

Just several lines of code are required to initialize database and there is no need to create a subclass of helper like in OrmLite or pure Android SQLite approach. Initialization is illustrated in listing 19.

```
// Creating helper instance
helper = new DaoMaster.DevOpenHelper(this, "messages-db",
null);
// Getting database instance with read/write rights
db = helper.getWritableDatabase();
// Creating DaoMaster object to manage database
daoMaster = new DaoMaster(db);
// Initialize session
daoSession = daoMaster.newSession();
// Initialize MessageDao object
MessageDao messageDao = daoSession.getMessageDao();
```

Listing 19. Initializing database and DAO objects using GreenDAO

GreenDAO requires schema to be defined for the code generator. This can be accomplished by adding entities to the schema as shown in listing 20. Entity properties and relations (“to-one” and “to-many”) can be created as follows:

```
// Schema and entity are created
Schema schema = new Schema(1, "com.qco.qco");
Entity message = schema.addEntity("Message");
note.addIdProperty();
note.addStringProperty("body").notNull();
note.addLongProperty("authorId").notNull();
note.addDateProperty("date");
// Generator produces boilerplate required for SQLite
new DaoGenerator().generateAll(schema,
"../GreenDao/src-gen");
```

Listing 20. Creating GreenDAO schema for code generator

Entities may inherit from super class, which cannot be entity as illustrated in listing 21. But there is no possibility to inherit another entity (polymorphic queries are not available).

```
messageA.setSuperclass("CommonMessageBehaviour");
```

Listing 21. Super class inheritance

It could be preferred to implement interfaces for common entity properties and behavior as described in listing 22.

```
messageA.implementsInterface("messageCommon");
messageB.implementsInterface("messageCommon");
```

Listing 22. Implementing common properties via interface

For example, if entities messageA and messageB have a common set of properties, these properties (including necessary getters and setters) can be placed in interface C.

4.3 Summary

Overview of storage solutions started off by investigating the simplest and least sophisticated storage method, the SharedPreferences class [15]. Key advantages and disadvantages were discovered while using a SharedPreferences solution in an application, and though the class is quite limited and allows to store primitive data types only, several important use cases exist.

Both SQLite [15] and Realm [19] databases were examined to understand drawbacks and benefits. SQLite is a mature database technology which is stable, efficient and has built-in support in most popular mobile operating systems, such as Android and iOS. Its source code has been open-souce and is published in a public domain. Unfortunately, despite of its popularity and transparent nature, being originally developed in 2000, it brings all burden and unneccessary complexity into modern mobile platforms. Data in SQLite is represented via tables consisting of rows and multiple columns, an approach which migrated from the server side into the embedded database solution. On the contrary, mobile platforms are object-oriented in nature and developers operate with objects, not rows in the table. In modern RESTful Web services most popular format is JSON – objects are delivered in response to the REST client. If pure SQLite solution is used for persistence, object data need to be converted into database rows manually via inconvenient helper methods. Realm and various ORMs can be an appropriate solution in this situation. Both allow to manipulate data as objects and make life easier for developer.

Realm offers significant boost in performance compared to pure SQLite and SQLite-based ORMs according to an open-source code generic benchmark written by Realm.io team available on Github [22]. Comparison graphs are listed in appendix 2. Compared to SQLite Realm additionally supports such data types as Date and byte array byte[] [19]. Realm allows to eliminate use of content providers (if no data have to be presented to external applications), content values, URI matchers, SQL builders, cursors and column interfaces. The ContentProvider [8] could be extremely useful in Android, however. For instance, it helps to share database data across applications and makes possible to use a Sync Adapter. There is a way to implement ContentProvider with Realm, but URI queries are handled in a different manner (it may be needed to construct cursors manually and for some cases to use matrix cursors). It is important to mention that Realm lacks stable support for database migrations and encryption (this functionality is still under development and has some issues listed in the Github repository of the library).
Realm is a complete persistence solution and it includes its own database core. On the other hand ORM solutions act like an intermediary interface and utilize SQLite database, which is already built in into Android. ORMs map Java objects to database tables. In this way an application can store, update, delete, and query for Java objects using a simple object oriented API.

If choice is made between most popular ORM solutions, GreenDAO is considered faster solution than ORMLite due to its code generation mechanism: “For the same given entity, greenDAO inserts and updates entities over 2 times faster, and loads entities 4.5 times faster for loading entities than ORMLite”, according to the official GreenDAO documentation [21]. Several top Android applications rely on GreenDAO (for example, Pinterest, which has more than 50 million installs). This shows library reliability in production. [24]

To summarize discussion about persistence solutions, last choice belongs to developer and each use case should be investigated. Right choice of storage solution may save time and help developer to focus on real problems and achieve better result.

5 Implementing REST Client for Android

5.1 Introduction

This section is dedicated to implementation of REST client for Android in startup company "Quikoo" and describes whole development process from early beginning (choosing back-end solution) until product is ready for the market (managing data on the client side). However, it is important to start from concept of idea itself to understand deeper technical problems and implementation details.

Main features of the Quikoo application are as following:

1. Plan time for the user and time for people he/she communicates with.
   Time planning is achieved by sending so called "quiks": reminders, requests to do something or invitations to an event (business meeting, cup of coffee, party) making sure person will not forget to do, or arrive at the venue on time.
2. Quikoo is a social platform.
   It is possible to send public "quiks" that will reach people from all the world through own contacts, and contacts of contacts (via forward feature). Public comments are also allowed for each "quik".
3. Quikoo is a user-friendly mobile application.
   It is easy to use: in two clicks user can create "quik" with media files, location and reach the whole world.

Technical details about the application are summarized as following:

1. All data received/sent should be cached in database.
   Application uses SQLite database to store data.
2. Geolocation services.
   Application can use fine location (via GPS signal) or coarse location (base station triangulation and Wi-Fi network positioning) to execute reminders. Both services are provided by Android Framework.
3. Fine reminder timer.
   Application loads and sets reminders (basically alarms on Android) automatically on every reboot and when each reminder is added into database.
4. Every Quikoo user has a registered status and his own profile on the server.
   This data is synchronized in order to allow users to send "quiks" to each other.
5. Every Quikoo user keeps registered status and own profile on the server.

This data is synchronized in order to allow users to send “quiks” to each other.

5.2 Backend Solution

There are several challenges an average mobile developer deals with:

- Time investment. It usually takes a fair amount of time to develop the front-end for mobile app and development of back-end increases the work to be done significantly.
- Skill investment. If developer is experienced in Google Android or Apple iOS, it does not mean that he/she has experience in back-end development. They are completely different technologies, so it takes a huge amount of time to learn them, and the company and developer might not have the time. Especially if certain time constraints exist, nobody can guarantee the quality of the final product.
- Problems of scalability. Due to nature of mobile applications developer never knows whether app become used by millions of users worldwide or not. So, developing back-end (or choosing back-end solution) so that it scales efficiently along with increasing usage is quite important.

Nowadays developers no longer need to develop their own back-end for every application. Several companies offer ready-made and highly configurable web back-ends that can be easily integrated into the product. Back-end as a service (BaaS) is a new cloud computing service offered to minimize the complexity and time taken by app developers to build their application. All these services provide a backend storage and other functions, which can be accessed from mobile app, usually using a compact Android / iOS library for integration simplicity via embedded SDK. Most of the companies offer free accounts with a wide set of features included as well as priced tiers for apps that need to scale up (which sometimes cost up to several thousand USD; for example, Parse.com asks 3700 USD for the highest tier which allows to handle 400 request per second simultaneously).

Because Quikoo company did not have sufficient resources (a startup in a state without investments, no clients yet), initial decision was made to use one of the promising open-source solutions on the market called BaasBox. BaasBox is an open source BaaS released under the Apache 2 license and is totally free to download, free to use and free to modify. BaasBox is a server that provides general back-end services for both mobile and web applications. [24]

The most important that BaasBox can be installed on any platform like Google Cloud Computing, Amazon Elastic Compute Cloud (EC2), dedicated server or VPS (Virtual Private Server), at any time, and then backend services can be managed using administrator control panel. In the project BaasBox solution was installed on the cheapest VPS hosting which costs 5 USD per month. VPS hosting has sufficient resources for alpha and beta testing stage when the number of users is low.

5.3 BaasBox Android SDK and Client Initialization

The SDK is distributed as a jar. Current version is 0.8.4 (16.11.2014). It can be downloaded from BaasBox site [24] and integrated by including library in the project's libs folder. Developer can also use gradle or maven to add library dependency in build.gradle file (in Android Studio): "compile 'com.baasbox:baasbox-android:0.8.4' ".

BaasBox client should always be initialized when application is started in onCreate() method of Application class (listing 23).

```
BaasBox.Builder b = new BaasBox.Builder(this);
client = b.setApiDomain("address")
.setAppCode("appcode")
// Used for push notifications
.setPushSenderIds("google sender id")
.init();
```

Listing 23. Initialization of BaasBox client

Most BaasBox REST resources are accessible through wrapper classes. Endpoints can be reached through asynchronous methods, that accept a general callback interface BaasHandler<T> to handle result (success/failed). It is possible to access endpoints using synchronous alternatives using the *Sync version of the methods. Results are always wrapped in BaasResult<T>, which represents the actual result or a failure. Synchronization methods are useful when requests should be done sequentially (not in parallel) and are used in SyncAdapter project's class as described in the section below.

All asynchronous requests are executed by a pool of threads and can be managed by RequestToken returned value. Tokens allow to suspend the assigned callback without interrupting the request itself, which allows to resume processing current request later. This is quite useful when callbacks are tied to the lifecycle of activities (for example, when images are being attached to message in application and user decides to cancel image uploading process – this situation also exists in Quikoo and is described in section below).

Some endpoints of the BaasBox API can be reached only via rest() and restSync() methods of SDK, because they were not yet implemented in current version of SDK. Documentation includes “To be implemented” marks for certain cases. Some rest endpoints have no direct equivalent in the API. Documentation says: “Using these methods an application can access these APIs while still enjoying the rest of the SDK features, such as concurrency and lifecycle management, caching, handling of the authentication” [24], so it means there is no need to worry about Session ID handling and other unnecessary things, which is helpful. For certain cases in Quikoo project it was much easier to use rest() and restSync() methods, because they allowed to make a query by submitting unlimited number of parameters appended as a string.

5.4 Building BaasAccessor Class with Client Methods

In the early beginning, the application was built using a simple, but incorrect approach. All REST client methods were placed in Fragment and Activity classes and did not follow Google IO 2010 patterns [6] at all. It made app code totally cumbersome and unstable, because methods were not bind to activity lifecycle properly. Among other consequences of the initially chosen approach were problems with data caching, synchronization and high data traffic load due to repeated requests. Geofences (geolocation reminders) and time reminders were disabled. It was enough for the start, but insufficient to build complex application with all predefined features.

Quikoo application is based on frequent synchronizations, so Pattern C seems to be the most appropriate for adoption (Activity <-> Content Provider <-> Sync Adapter) [6, 26]. In order to make use of SyncAdapter BaasAccessor class was written to accommodate all client methods (which were spread across application before). Methods were created to fetch “quiks” for current user (getAllReceived(),getQuicksForUser()), to receive all sent “quiks” (getSentQuiks()) and information about users from the contact list (getRegistered()). Methods are reached via instance of BaasAccessor class (illustrated in listing 24).

```
BaasAccessor accessor = new BaasAccessor();
// Get all “quiks” from server
ArrayList<Quik> quiks = accessor.getQuiksForUser();
```

Listing 24. BaasAccessor class used for backend operations

In order to understand how data is fetched from server, data scheme should be present in listing 25.

```
// Unique name of the object
{
  “id": "e80ff970-fdf5-48e6-850d-39b0279544f2",
// Sender’s phone number
  "creator": "79991111111",
// Text of reminder
  "reminderText": "Reminder Text",
  "latitude": 55.75291,
  "longitude": 37.59846,
// Time in unix format (long integer)
  “reminderTime”: 23535355
// Type: 2 (reminder), 1 (request), 0 (reminder)
  "quik_type": 2,
// Id of the file on server
  "attached_photos": [
    "0b4bacba-1693-4e63-9fe8-51ad3274f34e"
  ]
}
```

Listing 25. Data scheme for BaasBox

Every reminder is saved according to this format. It handles all data related to “quik” in “quiks” collection, but receivers are saved separately in “receivers” collection as illustrated in listing 26.

```
{
  "quikId": "e80ff970-fdf5-48e6-850d-39b0279544f2",
  "rcv": "375207737506",
  "acc": 1, // 1 - accepted, 2 - declined
  "quik_type": 2 // quik type
}
```

Listing 26. Item in “receivers” collection

Field “quik_type” is saved for every receiver (used to fetch queries by type). “quikId” is a string key which corresponds to “id” field of the parent “quik”. Information about every registered user is saved in “rg” collection and has photo attached (identified by unique id stored on the server). Data scheme for “rg” collection is shown in listing 27.

```
{
// User number
  "n": "74952051986",
// Photo id attached
  "photoId": "b1a450e6-98db-4501-86a2-8d76a9c35f47"
}
```

Listing 27. Data scheme for every registered user

BaasAccessor class is also responsible for sending push messages to recipients.

```
final JsonObject message = new JsonObject()
.put("quik_type", quik_type).put("quikId", quikDocId)
.put("reminderText", reminderText).put("sender",
BaasUser.current().getName());
if (reminderTime!=null)
message.put("reminderTime", reminderTime);
if ((latitude!=null) && (longitude!=null)) {
message.put("latitude", latitude).put("longitude",
longitude
}
if (attached_photos.size()!=0)
message.put("attached_photos", attached_photos);
BaasBox.messagingService().newMessage()
.profiles(BaasCloudMessagingService.DEFAULT_PROFILE)
.text(message.toString()).to(users)
.send(new BaasHandler<Void>() {
@Override
public void handle(BaasResult<Void> result) {
```

```
// handle the result
if (result.isSuccess())
Log.e("BaasAccessor", "Push sent successfully");
else
Log.e("BaasAccessor", result.error().toString());
}
});
```

Listing 28. Sending push message as JSON string

JSON document is sent as a string via message field of push notification using BaasBox.messagingService() method (used in listing 28). Push can be delivered to many users at once using one request to server.

When Sync Adapter is performing synchronization, it compares reminders data on the server and in SQLite database using simple rule: If incoming reminders are missing in local database, they are inserted using ContentProvider's "insert" method. UriMatcher instance is used to differentiate tables by content URI to make sure that incoming reminders would never mix up with sent reminders.

```
@Override
public Uri insert(Uri uri, ContentValues values) {
int uriType = sURIMatcher.match(uri);
SQLiteDatabase sqlDB = database.getWritableDatabase();
long id = 0;
String table;
switch (uriType) {
case REMINDERS_INCOMING:
table=TABLE_REMINDERS_INCOMING;
id = sqlDB.insert(table, null, values);
break;
case REMINDERS_SENT:
table=TABLE_REMINDERS_SENT;
id = sqlDB.insert(table, null, values);
break;
default:
throw new IllegalArgumentException("Unknown URI: " +
uri);
}
getContext().getContentResolver().notifyChange(uri,
null);
return Uri.parse(table + "/" + id);
}
```

Listing 29. Insert method of Content provider

If incoming reminder is modified by a local user, it will be saved in the database with modifiedByCurrentUser flag (every user can choose different reminder time and location according to specifications). During synchronization process, reminder document on the server, which corresponds to the current user, is updated using BaasAccessor (HTTP PUT implementation in BaasBox SDK). Reminder resources which carry user information are stored in separate collection called ”receivers” and common reminder information is saved in ”quiks”.

An essential part of the application design is an AlarmHelper class and its method ContentValues populateContent(Quik quik). It allows to rapidly store a set of values which ContentProvider can use in one of the predefined methods (for instance, insert, in listing 30)

```
// Generate ContentValues from Quik object
ContentValues cv=AlarmDBHelper.populateContent(quik);
provider.insert(Uri.parse("content://" + QuikooCon-
tentProvider.AUTHORITY + "/" + QuikooContentPro-
vider.TABLE_REMINDERS_INCOMING), cv);
```

Listing 30. Generating ContentValues object

Opposite operation is also necessary in order to construct an object from cursor data. All necessary fields can be assigned using a pattern as described in listing 31.

```
// An example of getting value from cursor
fieldname = cursor.getString(cursor.getColumn-
Index(column_name));
// Getting reminder time via cursor method
quik.reminderTime = cursor.getLong(c.getColumn-
Index(AlarmContract.COLUMN_NAME_REMINDER_TIME));
```

Listing 31. Pattern of getting value from cursor

AlarmContract.COLUMN_NAME_REMINDER_TIME is a static final string used to store name of the column for reminder time. Every column has a unique name and its own data type (for example, TEXT or INTEGER). SyncAdapter is configured to update all records automatically every six hours (to maintain service integrity). Manual synchronization is triggered when application is started for the first time, on Push Notification (single resource database insertion) and when new ”quik” is created (client -> server synchronization). Fragments, which hold list views in application Activity, register Cursor Loader callbacks. When data in the database is changed (records inserted / modified / deleted), fragment is notified about changes and ListView updates with new data.

```
// loader is initialized in onCreate method of Frag-
ment
getActivity().getSupportLoaderManager().initLoader
(LOADER_ID, null,this);
@Override
public Loader<Cursor> onCreateLoader
(int id, Bundle args) {
// create the Cursor that will take care of the data
being displayed
Uri uri = QuikooContentProvider.CONTENT_URI_INCOMING;
return new CursorLoader(getActivity(), uri, null,
null, null, null);
}
@Override
public void onLoadFinished(Loader<Cursor> arg0, final
Cursor cursor) {
incomingCursorAdapter.swapCursor(cursor);
}
@Override
```

```
public void onLoaderReset(Loader<Cursor> arg0) {
incomingCursorAdapter.changeCursor(null);
}
```

Listing 32. Using cursor loader inside Activity

Every single change in content provider triggers immediate change in cursor loader (listing 32), so it is possible to see even how records are sequentially inserted into database.

5.5 New API: Challenges and Solutions

After several months of development, it was decided to build company own API and host PHP server (Yii Framework) on Azure Cloud Hosting. That happened because BaasBox solution was not improved fast enough as it was intended in the beginning of the project and certain limitations were met, which did not allow placing all necessary logic on the server side. Moreover, due to the server-side limitations client code had become cumbersome already and made it impossible to adopt further significant improvements and new features. Development of the server side code was dedicated to the team from the partner company. After a few weeks of collaboration, a working API and documentation were received and implementation of changes in the client application began. Latest version of the product built in March 2015 is illustrated in screenshots, in figure 6.

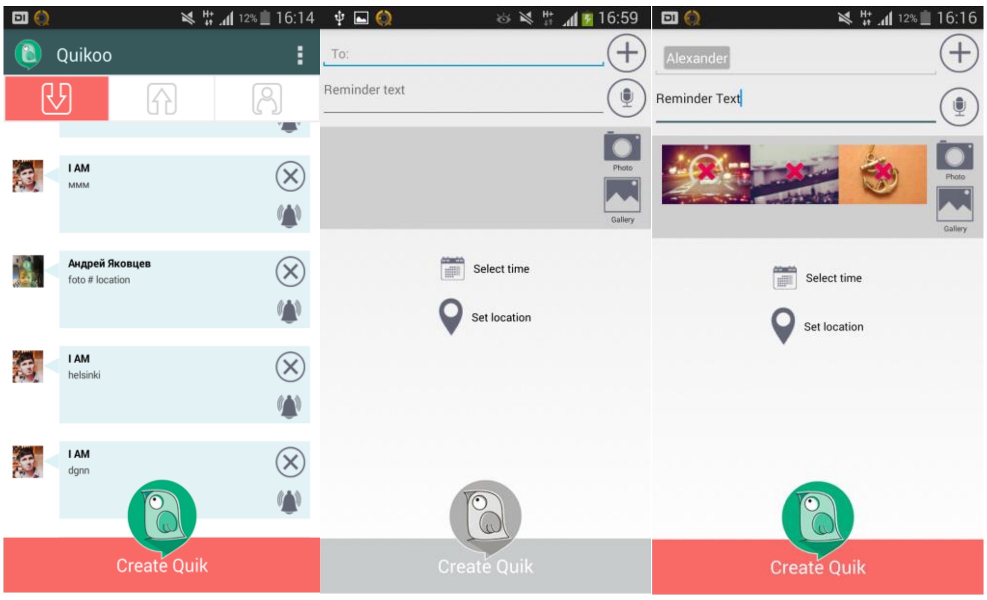


Figure 6. Application screens (left to right): “Incoming reminders”, “new reminder” dialog not filled with data and “new reminder” dialog with data attached

Initially was decided to drop BaasBox library from the project (it works only with BaasBox API) and search for an appropriate replacement. The research done in chapter 3 and 4 led to the conclusion that the best solution could be a pair of Retrofit [14] and Realm [19]. Codes listed below demonstrate that it is possible to build complex persistence solution in the most efficient way using these libraries.

Realm key feature is an ability to assign object received in response from Retrofit into RealmObject. Retrofit itself helps to build simple lightweight backend class integrated with API methods. An API is represented as an interface with several methods. API interface is illustrated in listing 33.

```
public interface API {
@FormUrlEncoded
@POST("/getMyQuicks")
GetAllQuiksResult getAllQuiks(@Field("user_id") String
userId, @Field("token") String token);

@FormUrlEncoded
@POST("/assignedQuick")
GetAllQuiksResult getAssignedQuiks(@Field("user_id")
String userId, @Field("token") String token);
```

Listing 33. Implementing API interface

Retrofit uses annotations to build request configuration. @POST annotation is required, because illustrated methods are documented as HTTP POST in API. The built-in Retrofit methods are GET, PUT, POST, HEAD, and DELETE. @FormUrlEncoded means that the request body will use form URL encoding and fields should be declared as parameters with @Field annotations.

```
Gson gson = new GsonBuilder()
.setExclusionStrategies(new ExclusionStrategy() {
@Override
public boolean shouldSkipField(FieldAttributes f) {
return f.getDeclaringClass()
.equals(RealmObject.class);
}
```

```
        })
        .create();
```

Listing 34. Configuring exclusion strategy for GSON

In order to bind objects received in response automatically, exclusion strategies should be set for GSON (listing 34) and proper GsonConverter must be defined for Retrofit adapter as shown in listing 35.

```
        // Configure Retrofit to use the proper GSON converter
        restAdapter = new RestAdapter.Builder()
        .setEndpoint(API_URL)
        .setConverter(new GsonConverter(gson))
        // Setting LogLevel to FULL allows to catch bugs
        .setLogLevel(RestAdapter.LogLevel.FULL)
        .setLog(new RestAdapter.Log() {
        @Override
        public void log(String msg) {
        // Log tag name is defined here
        Log.e("RETROFIT", msg);
        }
        })
        .build();
```

Listing 35. Creating an instance of Retrofit adapter

Finally, instance of API interface should be created (listing 36) in order to execute remote REST methods. All methods of an API interface run on the same thread (blocking behavior) and the best place for the code related to backend is SyncAdapter [9] (in this project it was left for compatibility) or Service running on background thread (for example, IntentService). However, there is a possibility to use callback for asynchronous method execution. In that particular case, callback object should be specified in parameters additionally [14].

```
        // Create an instance of our API interface.
        api = restAdapter.create(API.class);
```

Listing 36. Creating an instance of API interface

Persistence is achieved via Realm library as shown in listing 37. Changes are saved with commitTransaction method.

```
Realm realm = Realm.getInstance(context);
// Quick is a type of RealmObject
List<Quick> quiks = api.getQuiksForUser();
// Incoming quiks from remote database
List<Quick> quiksSent = api.getAssignedQuicks();
realm.beginTransaction();
List<Quick> realmQuicksIncoming =
realm.copyToRealm(quiks);
List<Quick> realmQuicksSent =
realm.copyToRealm(quiksSent);
realm.commitTransaction();
realm.close();
```

Listing 37. Handling persistence with Realm

Realm perfectly accepts an object, received from Retrofit and converted by GSON, library component included with Retrofit. No extra action is required. Developer gets full benefits from Retrofit + Realm pair: code is concise; there is no need to write complicated structures (as it was with pure SQLite + BaasBox). If extra methods need to be added later, that could be done in a few minutes. This approach saves time and really helps to focus on other important issues such as UI and new features. Modularity of an application raises and that makes possible to modify behavior in future releases without the need to rewrite huge blocks of code.

6 Conclusions

Several months have gone since work on the thesis began. The project work has passed through several stages of development, and underwent a huge number of improvements and optimizations. It was beneficial to learn about existing technologies used for networking and persistence; that made it possible to achieve the necessary skills to build an implementation of REST client for production. Essential knowledge was gathered about RESTful Web services, which allowed building a better, more responsive REST client. There are many ways to do the same task in Android and that is applicable also to interaction with RESTful Web service. Google provided the suggested implementation patterns, although the last choice belongs to the developer.

Project specifications should be studied and discussed beforehand to clearly understand the drawbacks and advantages in each particular case. The most important issue in development of a mobile REST client is to build an application as modular as possible. That simplifies development process, makes future API upgrades less time-consuming and the code cleaner (garbage code is taken away). Libraries described in the thesis will continue to be tested thoroughly and used in the following projects. New development components required for networking and persistence are continuously tracked and are subject to research in later work.

References

1. Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. Irvine, CA: University of California; 2000.

2. Rodriguez Alex. RESTful Web services: The basics [online]. IBM; February 2015, URL: http://www.ibm.com/developerworks/library/ws-restful/

3. Fielding & Reschke. RFC 7231 [online]. Internet Engineering Task Force (IETF); June 2014,
URL: https://tools.ietf.org/html/rfc7231

4. Fielding & Reschke. RFC 7230 [online]. Internet Engineering Task Force (IETF); June 2014,
URL: https://tools.ietf.org/html/rfc7230

5. Richardson L., Ruby S. Restful Web Services. Sebastopol, CA: O'Reilly; 2012.

6. Virgil Dobjanschi. Developing Android REST Client Applications. Presentation at Google I/O 2010 [online]. San Francisco, CA: Google; 2010.
URL: https://dl.google.com/googleio/2010/
android-developing-RESTful-android-apps.pdf
Accessed 27 March 2015.

7. Mednieks Z., Dornin L., Meike B., Nakamura M. Programming Android. 2nd ed. Sebastopol, CA: O'Reilly; 2012.

8. Android Developer API Documentation: Content Providers [documentation online]. Mountain View, CA: Google; 2015.
URL: http://developer.android.com/guide/topics/providers/content-providers.html
Accessed 27 March 2015.

9. Android Developer API Documentation: Running a Sync Adapter [documentation online]. Mountain View, CA: Google; 2015.
URL: http://developer.android.com/training/sync-adapters/running-sync-adapter.html
Accessed 27 March 2015.

10. Android Developer API Documentation: HttpURLConnection class [documentation online]. Mountain View, CA: Google; 2015.
URL: http://developer.android.com/reference/java/net/HttpURLConnection.html
Accessed 27 March 2015.

11. Repository of the Open Source Android library: RoboSpice [software component]. Version 1.4.14. Source codes. Paris, France: Octo Technology; 2014.
URL: https://github.com/stephanenicolas/robospice
Accessed 27 March 2015.

12. Resource oriented REST client for Android: Datadroid [software component]. Commit 0553cada0d. San Francisco, CA: Nicolas Klein; 2014.
URL: https://github.com/foxykeep/DataDroid
Accessed 27 March 2015.

13. Resource oriented REST client for Android: RESTdroid [software component]. Version 0.8.2. Paris, France: PCréations; 2014.
URL: https://github.com/PCreations/RESTDroid
Accessed 27 March 2015.

14. Retrofit: A type-safe REST client for Android and Java [documentation online]. San Francisco, CA: Square Inc; 2014.
URL: http://square.github.io/retrofit/
Accessed 27 March 2015.

15. Android Developer API Documentation: Storage Options [documentation online]. Mountain View, CA: Google; 2015.
URL: http://developer.android.com/guide/topics/data/data-storage.html
Accessed 27 March 2015.

16. Wei J. Android Database Programming. Birmingham, UK: Packt publishing; 2012

17. Datatypes In SQLite Version 3 [documentation online]. Charlotte, NC: Hwaci; 2015.
URL: https://www.sqlite.org/datatype3.html
Accessed 27 March 2015.

18. Android Developer API Documentation: SQLiteOpenHelper class [documentation online]. Mountain View, CA: Google; 2015.
URL: http://developer.android.com/reference/android/database/sqlite/
SQLiteOpenHelper.html
Accessed 27 March 2015.

19. Realm: Mobile database [software component online]. Mountain View, CA: Y combinator, 2015.
URL: http://realm.io/
Accessed 27 March 2015.

20. OrmLite - Lightweight Object Relational Mapping (ORM) Java Package [software component]. Open-Source community; 2015.
URL: http://ormlite.com/
Accessed 27 March 2015.

21. GreenDAO – Android ORM for SQLite [software component]. München, Germany: Greenrobot; 2015.
URL: http://greendao-orm.com/
Accessed 27 March 2015.

22. Realm performance comparison tests. Open-Source project. [online]. Mountain View, CA: Y combinator; 2015.
URL: https://github.com/realm/realm-java/tree/rw-performance-comparison/tests
Accessed 27 March 2015.

23. http://www.appbrain.com/stats/libraries/details/greendao/greendao

24. Baasbox: the open source backend [online]. Rome, Italy: BaasBox S.R.L.; 2015
URL: http://baasbox.com
Accessed 27 March 2015.

25. Realm for Android [online]. Mountain View, CA: Y combinator, 2014.
URL: http://realm.io/news/realm-for-android/#realm-for-android
Accessed 27 March 2015.

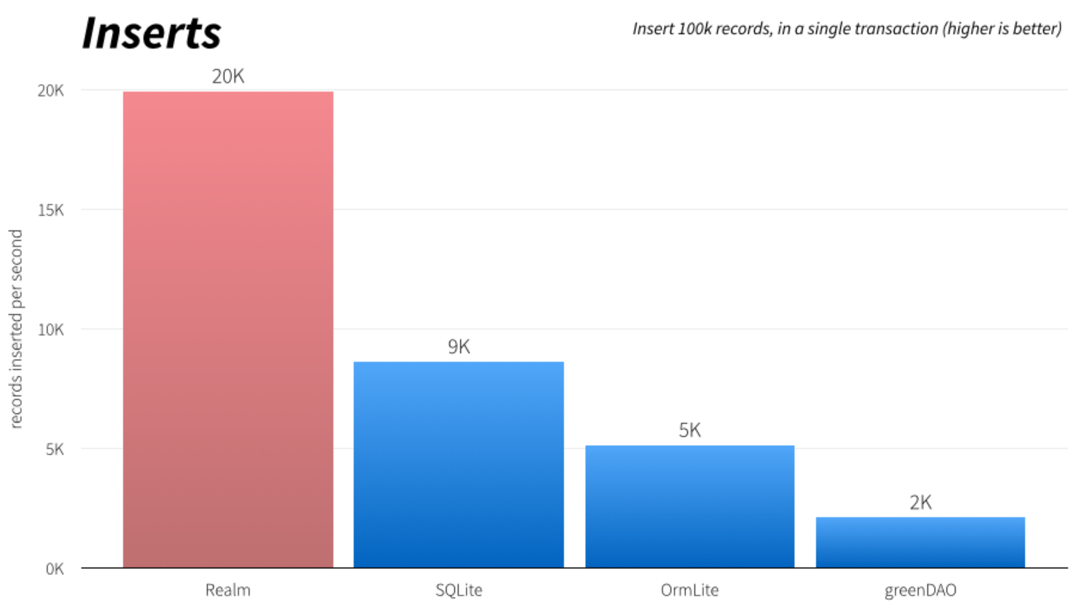
Appendix 1. SQLite Example

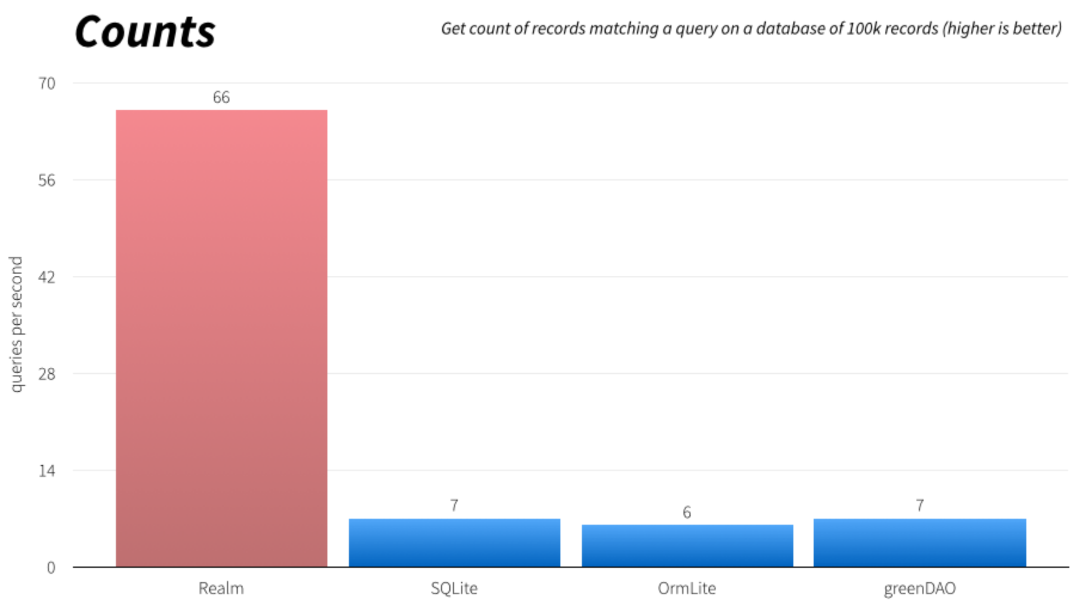
```
// INIT SQLITE HELPER
SQLiteHelper sqh = new SQLiteHelper(this);
// RETRIEVE A READABLE AND WRITEABLE DATABASE
SQLiteDatabase sqdb = sqh.getWritableDatabase();
// METHOD #1: INSERT USING CONTENTVALUES CLASS
ContentValues cv = new ContentValues();
cv.put(SQLiteHelper.NAME, "User Name");
// CALL INSERT METHOD
sqdb.insert(SQLiteHelper.TABLE_NAME, SQLiteHelper.NAME, cv);
// METHOD #2: INSERT USING SQL QUERY
String insertQuery = "INSERT INTO " + SQLiteHelper.TABLE_NAME + "
(" + SQLiteHelper.NAME + ") VALUES ('User Name')";
sqdb.execSQL(insertQuery);
// METHOD #1: QUERY USING WRAPPER METHOD
Cursor c = sqdb.query(SQLiteHelper.TABLE_NAME, new String[] {
SQLiteHelper.UID, SQLiteHelper.NAME
},           null, null, null, null, null);
while (c.moveToNext()) {
// GET COLUMN INDICES + VALUES OF THOSE COLUMNS
int id = c.getInt(c.getColumnIndex(SQLiteHelper.UID));
String name = c.getString(c.getColumnIndex(SQLiteHelper.NAME));
Log.i("LOG_TAG", "ROW " + id + " HAS NAME " + name);
}
c.close();
// METHOD #2: QUERY USING SQL SELECT QUERY
String query = "SELECT " + SQLiteHelper.UID + ", " +
SQLiteHelper.NAME + " FROM " + SQLiteHelper.TABLE_NAME;
Cursor c2 = sqdb.rawQuery(query, null);
while (c2.moveToNext()) {
int id = c2.getInt(c2.getColumnIndex(SQLiteHelper.UID));
String name = c2.getString(c2.getColumnIndex(SQLite-
Helper.NAME));
Log.i("LOG_TAG", "ROW " + id + " HAS NAME " + name);
}
c2.close();
// CLOSE DATABASE CONNECTIONS
```

```
sqdb.close();
sqh.close();
}
```

Appendix 2. Android SQLite ORMs and Realm Comparison Graphs

Tests run on an Galaxy S3, using the latest available version of each library as of Sept 28, 2014 (figure 7).





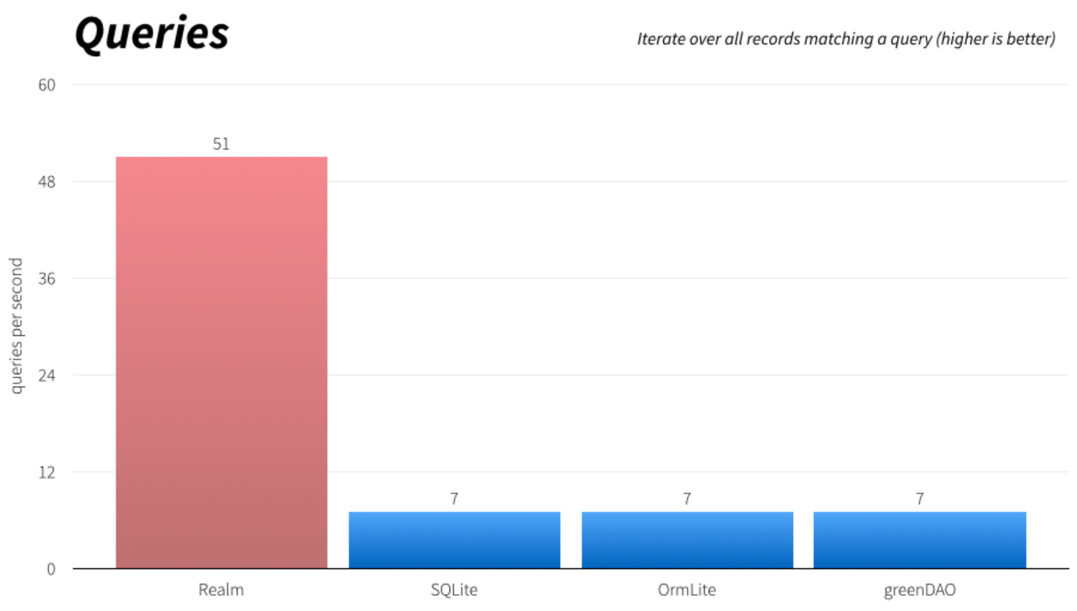


Figure 7. ORM, SQLite and Realm comparison [25]

Appendix 3. Source Code (Quikoo Project Based on BaasBox)

Sample method to send a reminder to the serve with a push notification:

```
public HashMap<String, Boolean> sendQuik(Context context, Quik quik, ArrayList<String> numbers) {
        BaasDocument note = new BaasDocument("Quiks");

        note.put("creator", BaasUser.current().getName());
        note.put("reminderText", quik.shortDescription);
        if (quik.latitude!=null)
            note.put("latitude",quik.latitude);
        if (quik.longitude!=null)
            note.put("longitude",quik.longitude);

        note.put("Quik_type", 2); //invitation = 0, request = 1, reminder = 2, contest = 4
        if (quik.reminderTime!=null)
            note.put("reminderTime", quik.reminderTime);

        String QuikId = null;
        Log.d("QuikAsyncTask", "1");
        PhoneNumberUtil phoneUtil = PhoneNumberUtil.getInstance();
        TelephonyManager t = (TelephonyManager) context.getSystemService(Context.TELEPHONY_SER-
VICE);
        String country = t.getNetworkCountryIso();
        if (country != null)
            country = country.toUpperCase();

        String QuikDocId = null;

        JsonArray attached_photos = new JsonArray();
        for (Map.Entry<ImageToUploadObject, RequestToken> entry : AppClass.requestTokens.entrySet()) {
            ImageToUploadObject key = entry.getKey();
            RequestToken value = entry.getValue();
            if (key.fileIdOnServer != null) {
                attached_photos.add(key.fileIdOnServer);
            }
        }
        note.put("attached_photos", attached_photos);

        if (QuikId == null) {
            // Let's save Quik first
            BaasResult<BaasDocument> QuikDocResult = note.saveSync(SaveMode.IGNORE_VERSION);
            //ArrayList<String> QuikIdList = new ArrayList<String>();
            Log.d("QuikAsyncTask", "2");
            try {
                if (QuikDocResult.isSuccess()) {
                    QuikDocId = QuikDocResult.get().getId().toString();
                    //QuikIdList.add(QuikDocId);
                    //Allow all registered users to see this Quik
                    BaasResult<BaasDocument> QuikDocFetchedResult = BaasDocument.fetchSync("Quiks",
QuikDocId);

                    if (QuikDocFetchedResult.isSuccess()) {
                        BaasDocument QuikDocFetched = QuikDocFetchedResult.value();

                        BaasResult<Void> grantResult = QuikDocFetched.grantAllSync(Grant.ALL, Role.REGIS-
TERED);

                        if (grantResult.isSuccess()) {

                        } else if (grantResult.isFailed()) {

                        }
                    } else if (QuikDocFetchedResult.isFailed()) {

                    }
                } else if (QuikDocResult.isFailed()) {

                }

            } catch (BaasException e) {
                e.printStackTrace();
            }
        } else {
            QuikDocId = QuikId;
        }

        Log.d("QuikAsyncTask", "3");

        ArrayList<BaasDocument> receivers = new ArrayList<BaasDocument>();

        HashMap<String, Boolean> results = new HashMap<String, Boolean>();

        //ArrayList<BaasUser> users = new ArrayList<BaasUser>();
        BaasUser[] users = new BaasUser[numbers.size()];
        int i = 0;
        for (String number : numbers) {

            String phoneNumberString = number.replaceAll("[\\D]", "");

            users[i] = BaasUser.withUserName(phoneNumberString);

            BaasDocument receiver = new BaasDocument("receivers");
            receiver.put("QuikId", QuikDocId);
            receiver.put("rcv", phoneNumberString);
            receiver.put("acc", 1);
            receiver.put("Quik_type", quik.Quik_type);
            receivers.add(receiver);

            i++;
        }
        Log.d("QuikAsyncTask", "4");
        ArrayList<QuikReceiverObject> receiverObjects = new ArrayList<QuikReceiverObject>();

        // Save all Quik receivers one by one
        for (BaasDocument receiver : receivers) {
            BaasResult<BaasDocument> rcvDocResult = receiver.saveSync(SaveMode.IGNORE_VERSION);
            try {
                if (rcvDocResult.isSuccess()) {
                    String rcvDocId = rcvDocResult.get().getId().toString();
                    receiverObjects.add(new QuikReceiverObject(rcvDocId, receiver.getString("QuikId"), re-
ceiver.getString("rcv"), 1, 1));
                    //Allow all registered users to see this Quik
                    BaasResult<BaasDocument> rcvDocFetchedResult = BaasDocument.fetchSync("receivers",
rcvDocId);

                    if (rcvDocFetchedResult.isSuccess()) {
                        BaasDocument rcvDocFetched = rcvDocFetchedResult.value();
                        BaasResult<Void> grantResult = rcvDocFetched.grantAllSync(Grant.ALL, Role.REGIS-
TERED);
                        if (grantResult.isSuccess()) {

                        } else if (grantResult.isFailed()) {

                        }
                    } else if (rcvDocFetchedResult.isFailed()) {

                    }
                } else if (rcvDocResult.isFailed()) {

                }

            } catch (BaasException e) {
                e.printStackTrace();
            }
        }

        for (QuikReceiverObject receiver : receiverObjects) {
            final JsonObject mess = new JsonObject()
                    .put("Quik_type", quik.Quik_type)
                    .put("QuikId", QuikDocId)
                    .put("reminderText", quik.shortDescription)
                    .put("sender", BaasUser.current().getName())
                    .put("currentQuikUserDocId", receiver.id)
                    .put("version", 1)
                    .put("version_receiver", receiver.version);
            if (quik.reminderTime != null)
                mess.put("reminderTime", quik.reminderTime);

            if ((quik.latitude != null) && (quik.longitude != null)) {
                mess
                        .put("latitude", quik.latitude)
                        .put("longitude", quik.longitude);
            }

            Log.e("Size of attached_photos array: ", attached_photos.size() + " " + attached_photos.toString());

            if (attached_photos.size() != 0)
                mess.put("attached_photos", attached_photos);

            BaasBox.messagingService()
                    .newMessage()
                    .profiles(BaasCloudMessagingService.DEFAULT_PROFILE)
                            //.extra(mess)
                    .text(mess.toString())//mess.getString("reminderText"))
                    .to(users)
                    .send(new BaasHandler<Void>() {
```

```
                    @Override
                    public void handle(BaasResult<Void> result) {
                        // handle the result
                        if (result.isSuccess())
                            Log.e("SendQuikHugeAsyncTask", "Push sent successfully");
                        else
                            Log.e("SendQuikHugeAsyncTask push error: ", result.error().toString());
                    }
                });
    }

    return results;
}
```

Update method of application ContentProvider class:

```
@Override
 public int update(Uri uri, ContentValues values, String selection,
              String[] selectionArgs) {

    int uriType = sURIMatcher.match(uri);
    SQLiteDatabase sqlDB = database.getWritableDatabase();
    int rowsUpdated = 0;
    switch (uriType) {
      case REMINDERS_INCOMING:
        rowsUpdated = sqlDB.update(TABLE_REMINDERS_INCOMING,
                values,
                selection,
                selectionArgs);
        break;
      case REMINDERS_SENT:
        rowsUpdated = sqlDB.update(TABLE_REMINDERS_SENT,
                values,
                selection,
                selectionArgs);
        break;
      case REMINDER_ID_INCOMING:
        String id = uri.getLastPathSegment();
        if (TextUtils.isEmpty(selection)) {
          String where = AlarmContract.Alarm.COLUMN_NAME_RIR_ID + "=?";
          String[] whereArgs = new String[] {String.valueOf(id)};
          rowsUpdated = sqlDB.update(TABLE_REMINDERS_INCOMING,
                  values,
                  AlarmContract.Alarm.COLUMN_NAME_RIR_ID + "=" + where,
                  whereArgs);
        } else {
          rowsUpdated = sqlDB.update(TABLE_REMINDERS_INCOMING,
                  values,
                  AlarmContract.Alarm.COLUMN_NAME_RIR_ID + "=" + id
                          + " and "
                          + selection,
                  selectionArgs);
        }
        break;
      case REMINDER_ID_SENT:
        String id2 = uri.getLastPathSegment();
        if (TextUtils.isEmpty(selection)) {
          String where = AlarmContract.Alarm.COLUMN_NAME_RIR_ID + "=?";
          String[] whereArgs = new String[] {String.valueOf(id2)};
          rowsUpdated = sqlDB.update(TABLE_REMINDERS_SENT,
                  values,
                  AlarmContract.Alarm.COLUMN_NAME_RIR_ID + "=" + where,
                  whereArgs);
        } else {
          rowsUpdated = sqlDB.update(AlarmContract.Alarm.TABLE_REMINDERS_SENT,
                  values,
                  AlarmContract.Alarm.COLUMN_NAME_RIR_ID + "=" + id2
                          + " and "
                          + selection,
                  selectionArgs);
        }
        break;

      default:
        throw new IllegalArgumentException("Unknown URI: " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return rowsUpdated;
 }
```