DE GRUYTER OPEN

HUNGARIAN JOURNAL OF INDUSTRY AND CHEMISTRY
Vol. 44(2) pp. 93–98 (2016)
hjic.mk.uni-pannon.hu
DOI: 10.1515/hjic-2016-0011

HUNGARIAN JOURNAL OF INDUSTRY AND CHEMISTRY
HJIC

MOBILE DATA SYNCHRONIZATION METHODS

MIKLÓS PÁL AND GÁBOR LÁNER

Capture IT Solutions and Consulting, Záhony u. 7 Building B, Budapest, 1031, HUNGARY

The study introduces and compares the performance of a classical and two innovative mobile data synchronization methods. A customized test environment will be created for every selected method researched. Performance, stability, and other measurement results will be produced from these environments, which will be the major outcome of the study.

Keywords: mobile synchronization, comparison, performance measurement, data optimization, slow network connections

1. Introduction

The main goal of Mobile Workforce Management software is to make fieldwork easier and more efficient. This requires all necessary data to be available and in sync with the server databases. This is why the synchronization module is one of the most important parts of Mobile Workforce Management software. During the fieldwork, there are various data connection conditions that the application needs to be conformed with. There are several data objects, e.g. task, user, client, etc., and multipart objects, e.g. document files, images to be synchronized. The usability of an application highly depends on the efficiency of this synchronization.

The main problem that synchronization needs to solve is to transfer data between the devices and the data source. The solution begins with the data source and through the communication channel ends with the saved data on the device. Three potential implementation methods will be introduced and compared in this article.

2. Experimental

All three considered solutions have different technical backgrounds. The first one is a SOAP (Simple Object Access Protocol) web service implementation based on XML communication. This is the classical way to transfer data between two different platforms. The protocol originally was designed for Microsoft in 1998. The XML-based web service is a widespread solution because it is platform independent and built on industry-wide standards. It was one of the first ways to build service-oriented, modular architectures using smaller applications is by the communication of web services instead of robust, monolithic systems.

As the technology was mainly used in enterprise environments, and has been in use for more than 15 years, naturally it consists of an antiquated approach compared to modern, lightweight services. While the technology is based on standard HTTP protocols, the requests and responses travel in objects called envelopes. All of the envelopes have a header and a body part, where the body contains the actual payload, the data that is the main reason of communication. The format of the envelopes is strict, and furthermore, there are several encoding, formatting, and parsing standards that have been created since the birth of the protocol. Unfortunately, not all implementations are compatible, it is easy to create a server that cannot digest the client's request, while both endpoints use valid but different soap formats, even though the raw format of the envelope is an easily readable XML file.

The strict format has both benefits and inconveniences. When the service at the server is ready, it is easy to generate a unique description XML, called WSDL (Web Services Description Language). This file not only helps the development of the client side, but there are several tools available that can generate almost all of the client-side code, that can be used to connect to the service. Overall, it is a strict, old fashioned, but really reliable way to approach services in the modern mobile world.

The second tested solution is the RESTful (Representational State Transfer) service in the architecture of microservices. REST is also known as RESTful architectural design, and was represented in 2000 by Roy Thomas Fielding in his dissertation at the University of California, Irvine [1]. REST has become the main architectural design for web and mobile development over the last few years. According to 'ProgrammableWeb', 69% of the newly created APIs were using REST while only 22% were using SOAP in 2014.

*Correspondence: www.capture.hu

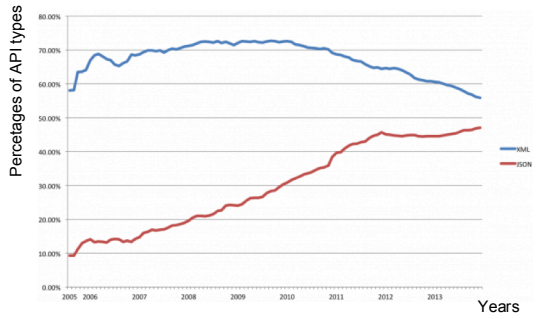

Figure 1. Percentage of APIs added to the ProgrammableWeb directory based on communication types over the years [2].

There are many advantages of this architectural design. The first is its solid performance, due to the high level approach of the solution. It typically communicates over HTTP (Hypertext Transfer Protocol) using HTTP verbs like GET, POST and PUT. REST supports more message formats, e.g. XML, CSV (comma-separated values), JSON (JavaScript Object Notation), etc. The primary communication format is JSON, which is structured text data type. This message format requires significantly less metadata than the XML format, thus greatly increasing the network efficiency as the valuable data can fit in smaller network packages. The spread of the JSON message format is shown in *Fig.1*. The diagram shows the percentage of JSON *versus* XML message formats used by APIs in the 'Programmable Web Directory' between 2005 and 2013 [2]. Another great advantage of RESTful is its simplicity. It is easy to implement and maintain due to its structure. It clearly separates the client and server implementations.

Today's trends point in a direction where developers need to create highly availabile applications exhibiting high level of scalability that are ready to run in cloud environments. Microservice architecture is a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal. The most well-known microservice architecture users are Netflix and Amazon. Applications based on this architecture are easy to understand and modify because of the independent parts. Instead of a robust application which contains all functions, logics and millions lines of code, there are many separated services with a focused function. These applications or application modules are able to run on multiple copies of multiple machines which makes them highly scalable, available and capable of running in cloud environments.

Patterns in programming are reusable solutions to a problem occurring in a particular context. In the world of microservice architecture there are many patterns available to choose from. From the aspect of deployment there are two main patterns:

- *Multiple services per host* - There is one physical server with all services installed on it.

Table 1. Data structure used in tests with the field data types and typical contents.

Column	Type	Example
ID	numeric	
LA reference	numeric	476483
promoter	character	Vultron
street	character	STONEGATE ROAD
locality	character	MEANWOOD
works type	character	STANDARD
easting	numeric	428804
northing	numeric	437215
location	character	OPP STAINBECK AVENUE VULTRON DUCTING FOR MAINS CABLE TO
description	character	DISPLAY IN BUS SHELTER
works start date	date	19/03/2008
works end date	date	08/06/2013

- *Single service per host* - In this case, there is a standalone host for each service. The host could be a virtual machine or a container.

The communication methods between the clients and servers are described with the API gateway communication pattern. In this pattern the gateway is a service discovery between the client and server. This service is the single entry point. From the aspect of the database there are two main patterns. The shared database pattern uses only one database for all services. The database per service pattern uses a standalone database for every single service.

Finally, the third solution is a distributed NoSQL implementation of mobile data synchronization. Early versions of NoSQL databases have existed since the 1960s but the technology started to spread only in the twenty-first century. NoSQL in other words means non-relational database. One of the main benefits of these databases is the simplicity in design, because they store data in a key-value structure. The other main benefit is the horizontal scalability with the support of clustered environments and cloud infrastructure. These types of databases are mainly used in big data environments. All three applications have the same functionalities.

During the experimental three demonstration applications were created. One separated environment for each featured solution. All the applications have three main modules with the same functionalities: (i) database at the backend side to store test data and (ii) synchronization module to transfer data between the backend and mobile application.

Mobile applications exist with the capability to connect to the synchronization module and synchronize data to the mobile device. Additionally, there are some status checking and logging functionalities on this side. Every mobile application was created with its own mobile database to store synchronized records. The data source of solutions was tested, which contains up to 50 thousand historical roadwork items of data from 2011 until 2016. The structure and an example record of the database are shown in *Table 1*. Test data is stored in a single table with a sequence number as a primary key.

The architecture of SOAP web service implementation is shown in *Fig.2*. On the database side

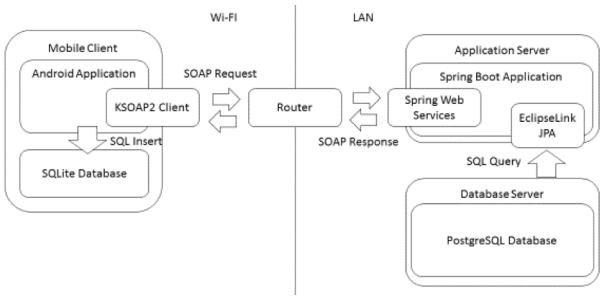

Figure 2. The SOAP based test application's logical architecture. The communication between the client and server component is based on XML / SOAP messages.

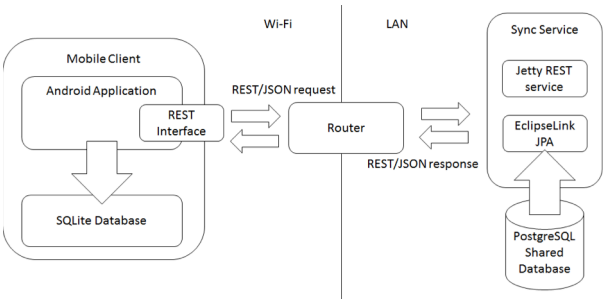

Figure 3. The REST based test application's logical architecture. The communication between the client and server component is based on JSON / REST messages.

there is a PostgreSQL server (version: "PostgreSQL 9.0.3, compiled by Visual C++ build 1500, 64-bit") installed. For the experimental, a separated database was created with pg_default tablespace and UTF-8 encoding. Inside the database test, tables were created in a public scheme.

The application server is a J2EE web application implemented with the SpringBoot framework, which builds a standalone runnable jar application that includes a WAR web application and also grants an embedded Apache Tomcat application server. This solution provides a monolithic architecture, which is widely used in the enterprise environment. The core framework of the application server is SpringBoot (version 1.3.6.-RELEASE), where the embedded Tomcat server version is 8.0.36. The web service itself is provided by Spring-WS (version 2.3.0.-RELEASE). The *getRoadworkListRequest* web service provides the main query about synchronization logic. This service performs a *select * from roadworks* query through the persistence layer and returns the whole list of the currently stored roadworks. The response is the XML representation of the data table presented in *Table 1*. As a persistence layer the application uses the EclipselLink JPA provider version 2.5.0.

The mobile application of SOAP implementation was built for the Android SDK version 24.0.0. Android does not offer any built-in library to handle SOAP calls. There are several third-party libraries to fill the gap, but one could not be found that could be a fully satisfactory solution to our problem. In the tests, the kSOAP2 (version 3.6.1) implementation was used that also has some very uncomfortable limitation, but during the test it was working reliably. The synchronized data is saved into the SQLite database in the mobile device. SQLite is the built-in Android database that offers a relational database with functionalities to access and store data.

The architecture of the RESTful service implementation is shown in *Fig.3*. This architecture is a typical microservice architecture where the service itself implements the synchronization functions. The mobile application implements the mobile-device functions like synchronization calls and status reports. A shared database pattern was used by sync service, which means the database used by synchronization service is a database used by other services, too. The same PostgreSQL database was used at the database level in the REST implementation as used before for the SOAP test application. Thus, the database version and database configuration were the same.

The service is implemented as a standalone Java application supported by Jetty (version 9.2.1.v20140609). Jetty provides a lightweight embeddable web server and it has support for REST APIs of Web Socket. These features make Jetty ideal to use in microservice architecture. Sync Service provides a REST API for mobile clients to conduct synchronization. The main API is the GET /rest/sync/roadworks HTTP/1.1; Content-Type: application/json service that responds with the whole list of roadworks as a JSON content type. The roadworks data is accessed with a full table selected from a database over an EclipseLink persistence layer. The same JPA provider (EclipseLink version 2.5.0) is used for this service as it is for the SOAP application.

The mobile application of the REST implementation was also built for the Android SDK version 24.0.0. However, Android has its own HTTP client provider, in this article Android-async-http (version: 1.4.9) was used for asynchronous HTTP client functionalities at the mobile application level. This is a well-featured and widely used library by top developers like Instagram and Pinterest [3]. The SQLite database was used to store synchronized data in the mobile device as well as in SOAP implementation. It was experienced during development that implementing REST API calls in the mobile environments is relatively easy to perform since it is a widely supported method of communication. Couchbase was used to build the NoSQL database because it offers a complete solution with a server-side database (Couchbase server), synchronization gateway and mobile-side database (Couchbase Lite). The architecture of this solution is shown in *Fig.4*.

The data layer is a Couchbase server (version 4.0.0-4051 Community Edition (build-4051)). The database has a single server node configured. The server node in Couchbase represents an instance of the database. In our test only one instance was run. In production environments, more instances are necessary to improve server availability. The node contains the physical data representation objects, called buckets. A new bucket was configured for the article as a Couchbase bucket type with 200 MB of memory allocated per node. The optimization of disk I/O

operations was set to default, which means the disk I/O priority is low for this bucket. In this article this is an issue, because there are no other buckets in use. The auto-compaction settings are also set to default which means auto-compaction should run if the fragmentation is above 30%.

The Sync Gateway (version 1.2.1 was installed) is located at the server side as a standalone application. This module implements database read / write functions and solution specific APIs to transfer data to and from mobile devices. It has a built-in versioning logic, which adds revision information to the documents stored in the server of the database and handles synchronization metadata like synchronization cycles and user data. The Sync Gateway is configured. It was created to setup the gateway to sync every document type without any user authorization. In this case every connected device synchronizes every document without restrictions between the client and server.

The mobile application of NoSQL implementation is similar to the previous solutions built for Android SDK version 24.0.0. The most significant difference here is the mobile database, which is Couchbase Lite (version 1.3.0). This is a mobile database created for Couchbase and Sync Gateway. It contains the mobile database engine, the mobile database handler and the synchronization interface implementation. The synchronization supports both push and pull requests with version checking so only modified documents are transferred during a call.

2.1. Samples and Measurements

Four types of measurements were performed during the experiment: speed test between server and devices for different numbers of datasets (small 1-10, medium 1,000-10,000 and a large number of records up to 50,000). Speed tests were performed during data transfer to and from the devices and with mixed directions. Speed and stability tests were performed using text and binary data types with a high amount of data to transfer. During the tests, all data packages and sizes of the packages were monitored, as well as the performance of mobile applications, synchronization gateways and databases.

3. Results and Analysis

3.1. Experiments

During the analysis an attempt was made to provide constant conditions with the following hardware infrastructure. An Asus K53S notebook with Intel® Core™ i7-2630QM CPU, 8 GB RAM and HDD WDC WD7500BPVT-80HXZ was used as the server to run the database, application server, synchronization service and synchronization gateway. A Samsung Galaxy SM-G935F (S7 Edge) smart phone with Android 6.0.1 (build number MMB29K.G935FXXS1APG2) was used as the mobile environment.

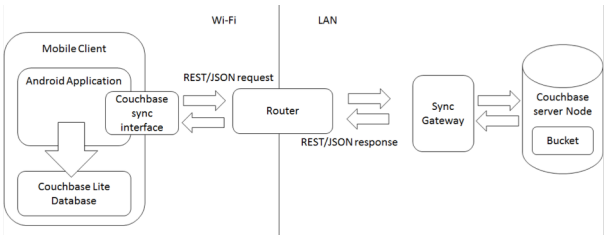

Figure 4. The CouchBase based test application's logical architecture. The communication between the client and server component is based on the database standard synchronization gateway.

Every test was run on the same local network. The network used a 100 Mb/s WIFI router. The server was connected to the router with a local area network (LAN) cable, the mobile device was connected *via* a WIFI network. This way the network speed during the experiment was constant.

During the experiment, six test rounds were run on the three different solutions. The amount of data was raised in every step from 1 row to 50,000 rows. One test round with 100,000 records was also planned, but the tests revealed the limitation of the mobile hardware, for around 30 MB of data, the response could not be parsed in one batch, mainly due to the lack of memory. To achieve realistic conclusions from the measurements, every step was repeated three times. Overall, a total of 54 tests were run.

Measurement results were collected using several methods. In SOAP and REST implementations, most of the information was collected from the mobile platform. Both applications were provided with a logger module that provided log entries in every main step of the synchronization. These steps were the following:

- synchronization initialized
- synchronization started (request was sent from the mobile device to the server)
- synchronization finished (the response came back from the server)
- parse start (when the mobile application started to process the response)
- parse done (when processing finished and all records from the response were saved in the database of the mobile client)

In the case of the NoSQL solution, the monitoring was a bit different because there was no way to write a custom logger module for the built-in processes. Fortunately, the Couchbase Sync Gateway provides a fine-grained log where nearly all equivalent steps can be found that we redefined for the previous tests.

The performance and mobile database monitoring was the other main part of the analysis. This part was the same for all solutions including NoSQL. The performance was monitored continuously with an Android debug tool while the database was monitored from the application with a status screen. It was expected that the SOAP Web Service implementation would be significantly slower than the RESTful and NoSQL solutions mainly because of the larger data packets transferred in XML format.

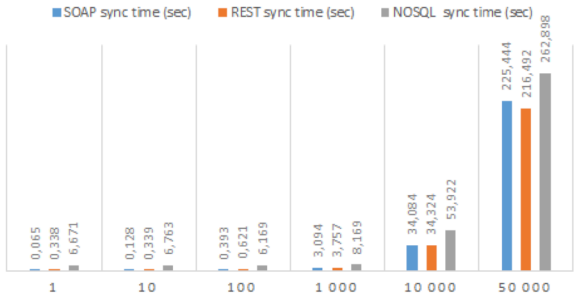

Figure 5. The synchronization time required for each test application with different record counts, in seconds.

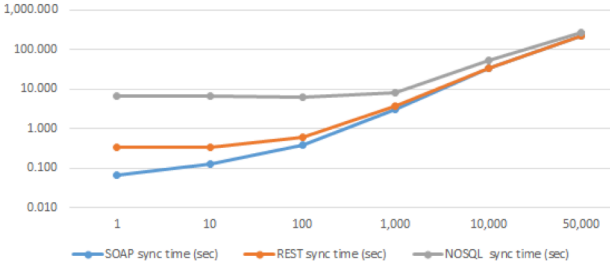

Figure 6. The synchronization time required for each test application with different record counts, in seconds, on logarithmic scale.

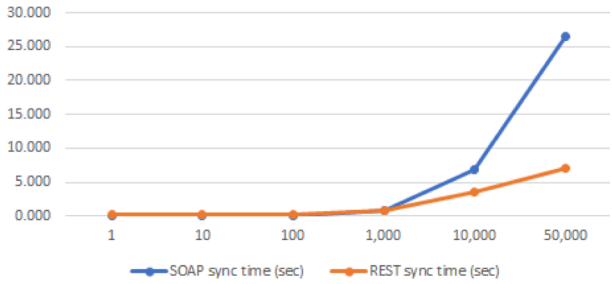

Figure 7. The synchronization time required for the SOAP and REST based test applications with different record counts, in seconds, without data parsing and persisting.

The measured results refuted these expectations. As *Fig.5* and *Table 2* show, the average sync times are not just nearly the same but with smaller data amount the SOAP is even faster than the REST. In this result, the NoSQL lags behind the other two solutions, but the sync time gets closer as the amount of data increases. The reason for this difference could be the additional versioning features of Couchbase. Furthermore, NoSQL is built for working with high amounts of data.

The synchronized data amount on the horizontal axis is increasing nearly logarithmically because of this, a logarithmic view of this result set (*Fig.6*) could yield a better understanding. *Fig.6* shows the key point is at 1,000 rows. Here is the point where all solutions start to converge into each other. From this point, increase in the sync time becomes more directly proportional to the increased in the data amount. By taking into consideration the result, numbers and sync time per row values in *Table 3*, is can be seen that the minimum value using SOAP is at 1,000 records, using REST the minimum value is at 10,000 and using NoSQL the minimum is above 50,000. The results above were

Table 2. The synchronization time required for each test application with different record counts, in seconds.

Number of rows	SOAP sync time (sec)	REST sync time (sec)	NoSQL sync time (sec)
1	0.065	0.338	6.671
10	0.128	0.339	6.763
100	0.393	0.621	6.169
1,000	3.094	3.757	8.169
10,000	34.084	34.324	53.922
50,000	225.444	216.492	262.898

Table 3. The time needed to synchronize one record for each test applications in different package sizes.

Number of rows	SOAP sync time (s/record)	REST sync time (s/record)	NoSQL sync time (s/record)
1	0.06533	0.33833	6.67100
10	0.01280	0.03390	0.67633
100	0.00393	0.00621	0.06169
1,000	0.00309	0.00376	0.00817
10,000	0.00341	0.00343	0.00539
50,000	0.00451	0.00433	0.00526

Table 4. The synchronization time required for the SOAP and REST based test applications with different record counts, in seconds, without data parsing and persisting.

Number of rows	SOAP sync time – without parsing (s)	REST sync time – without parsing (s)
1	0.044	0.278
10	0.090	0.239
100	0.134	0.243
1,000	0.815	0.764
10,000	6.842	3.589
50,000	26.540	7.020

Table 5. The size of the data packages using XML and JSON format, in kilobytes for different record counts.

Number of rows	SOAP XML size (KB)	REST JSON size (KB)
1	0.727	0.272
10	6.200	3.700
100	55.400	32.900
1,000	554.400	331.600
10,000	5,734.000	3,481.600
50,000	29,286.000	17,920.000

calculated using synchronization and data processing. If data processing is skipped, the results change as shown in *Table 4*.

Again, the key point here is the limit where the number of records is 1,000. After that point the synchronization using REST increases much faster. As shown in *Fig.7*, the SOAP sync time rises sharply while the REST sync time rises less rapidly. The size of the messages can only be monitored for the SOAP and REST implementations. The results of these measurements were the same as expected. Due to the strict data format, the XML structure requires larger amount of data transfer packages than JSON, as shown in *Table 5*. The logarithmic diagram in *Fig.8* shows that the increase in size is directly proportional to the number of transferred rows.

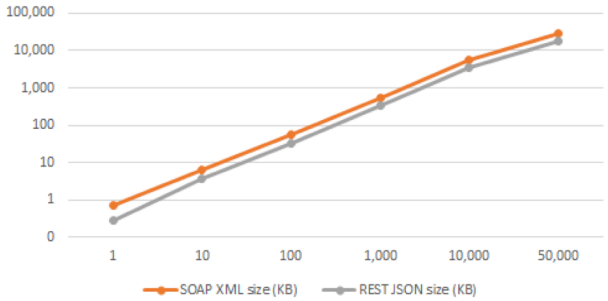

Figure 8. Comparison of the size of the data packages between the XML and JSON format on a logarithmic scale.

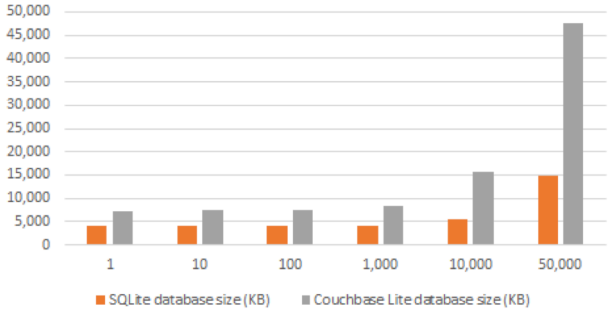

Figure 9. Comparison of the size of the databases between the SQLite and CouchBase databases.

The last measured value is the size of mobile database after data synchronization. The SOAP and REST implementations have the same database size because both of them used an SQLite database with the same data. This is the reason why only SQLite and Couchbase Lite databases were compared in *Table 6*. As shown in *Table 6* and *Fig.9*, the size of the Couchbase Lite database is much bigger than that of SQLite. The difference increases as the amount of data rises.

4. Conclusion

Finally, we need to state that SOAP performed surprisingly well during the experiment performance tests. The biggest limitation with regards to it is the minimal support in mobile development. Because in enterprise companies SOAP is still the most common technique this architecture is still popular.

More measurement data confirmed that there is a common key point where there is only a minimal difference between the selected solutions. This point is around 1,000 records per transaction, which is the point where it does not matter which solution is used. This is not the optimal point of performance for all solutions but it could be a good compromise.

Table 6. The size of the database on the client device using SQLite and CouchBase Lite databases for different record counts.

Number of rows	SQLite database size (KB)	Couchbase Lite database size (KB)
1	3,993.600	7,168.000
10	4,003.840	7,372.800
100	4,044.800	7,536.640
1,000	4,167.680	8,325.120
10,000	5,457.920	15,656.960
50,000	14,704.640	47,462.400

The best choice is the REST synchronization, if the goal is to quickly implement a customizable, reliable, scalable, and extendable, cloud-ready modern solution. Any amount of data is supported from small datasets to big data solutions. The bottleneck of this solution occurs during data processing.

The best choice is the SOAP synchronization if the goal is to create an enterprise-ready highly secure and auditable solution. This solution is not recommended for big data environments, but up to medium amounts of data, it could offer a real alternative to RESTful service. Working with SOAP has many limitations in mobile development.

NoSQL is the best choice for big data environments where a very large amount of data needs to be processed and there is a limited time for development.

Acknowledgement

We acknowledge the financial support of this work by the Hungarian State under the VKSZ_12-1-2013-0088 project.

REFERENCES

[1] Fielding, R.T.: Architectural styles and the design of network-based software architectures, Ph.D. Dissertation, University of California, Irvine, 2000

[2] DuVander, A.: JSON's eight year convergence with XML, 2013 www.programmableweb.com/news/jsons-eight-year-convergence-xml/2013/12/26

[3] Smith, J.: Android asynchronous HTTP client, 2016 loopj.com/android-async-http