Datalift

Un ascenseur pour les données

ANR Contint – ANR-10-CORD-009

D6.3 Mobile API for linked data

Coordinator: Jérôme David
With contributions from: Jérôme Euzenat (INRIA Exmo), Maria-Elena Roşoiu (INRIA Exmo)

Quality reviewer:	Luca Costabello (INRIA Wimmics) & Raphaël Troncy (Eurecom)
Reference:	Datalift/2011/D6.3/v1
Project:	Datalift ANR Contint ANR-10-CORD-009
Date:	June 21, 2012
Version:	1
State:	final
Destination:	public

Executive Summary

Mobile devices are now a main way to access data. Yet they do not offer any uniform way to deal with it.

Linked data provides a uniform interface for accessing linked data over the web.

We have developed an API in order to offer uniform access to data within mobile devices and across devices. It extends the Content Provider API designed to share data across Android applications. Finally, this API introduces a transparent URI dereferencing scheme allowing for exposing device content to others.

As a consequence, any application can access data as linked data without any a priori knowledge of the data source.

This deliverable is an extended version of [3].

Document Information

ANR Project Number	ANR Contint – ANR-10-CORD-009	**Acronym**	Datalift
Full Title	Un ascenseur pour les donnes		
Project URL	`http://www.datalift.org/`		
Document URL			

Deliverable	**Number**	6.3	**Title**	Mobile API for linked data
Work Package	**Number**	4	**Title**	Data interlinking

Date of Delivery	**Contractual**	M18	**Actual**	30-03-2012
Status	final		final ☒	
Nature	prototype ☒ report ☐ dissemination ☐			
Dissemination level	public ☒ consortium ☐			

Authors (Partner)	Jérôme Euzenat (INRIA Exmo), Maria-Elena Roşoiu (INRIA Exmo)			
Resp. Author	**Name**	Jérôme David	**E-mail**	Jerome.David@inria.fr
	Partner	INRIA Exmo		

Abstract (for dissemination)	This report presents a mobile API for manipulating linked data under the Android platform.
Keywords	mobile, API, linked data, content provider

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
20/05/2012	1	J. Euzenat	Initial version
07/06/2012	2	J. Euzenat	Accounted for feedback from Raphaël Troncy
20/06/2012	3	J. Euzenat	Accounted for feedback from Luca Costabello

Table of Contents

1. Introduction

Smartphones are becoming the main personal information repositories. Unfortunately, this information is stored in independent silos managed by applications and thus, it is difficult to share data across them. Nowadays, mobile operating systems, such as Android, deliver solutions in order to overcome this, but they are limited by the application database schemas that must be known beforehand.

The difficulty of sharing phone data at the web scale can be seen as another drawback. One can synchronize application data, such as the contacts or the agenda using a Google account. However, they are not generic solutions and there it is no mean to give access to data straight from the phone. The W3C Device API[1] cover this need across devices but they do offer specific APIs for specific applications and not a uniform and flexible access to linked data.

Our goal is to provide applications with a generic layer for data delivery in RDF. Using this solution, applications can exploit device information in an uniform way without knowing from the beginning application schemas. This information can also be exposed to the web and web information can be considered in the same uniform manner. For example, an application may be used as a personal assistant: when one would like to know which of his contacts will participate to an event, he can consult the calendar of all his contacts in order to retrieve the answer. For sure, according to the security settings of the corresponding contact, he may be allowed or not to access the calendar.

Moreover, we propose to do it along the linked data principles (use URIs, provide RDF, describe in ontologies, link to other sources).

The mobile device information can as well be accessed remotely, from any web browser, by any person who has been granted access to it. In this case, the device acts like a web server.

Some proposals have been made in that direction [6]. We presented a first version of the RDF content provider in [2]. This layer, built on top of the Android content provider, allowed to share application data inside the phone. In this paper, we extend the previous version by adding capabilities to access external RDF data and to share application data as linked data on the web.

We first describe the context in which the Android Platform stores its data, and how it can be extended in order to integrate RDF. Then, we present two applications that sustain its feasibility: the first one is an RDF browser that acts like a linked data client and allows the navigation through the device information, and the second one is an RDF server which exposes its information to the outside world. We continue to present the challenges raised by such applications and solutions we implemented for them. Finally, we conclude presenting future improvements and challenges in this field.

[1] `http://www.w3.org/2007/uwa/Activity.html`

2. The Android architecture

Android is a Linux-based operating system for mobile phones. It allows for developing applications in Java [7, 5].

Inside the Android system, each application runs in isolation from other ones. Android assigns to each application a different and unique user. Only this user is granted access to the application files. This allows one to take advantage of a secure environment, but prevents the exchange of data across applications. To overcome this drawback, Android provides the content provider mechanism.

2.1 Services and Intents

Android is built around different kinds of services that are provided by an application[1]:

- Activities are user interaction modalities (an application panel, a chooser, an alert);
- Services are processing tasks;
- Broadcast receivers are filters which react to events;
- Content providers expose some data of the application to other ones.

An application is build around zero or more of each kind of services. Applications and application components communicate through messages called "intent". They are defined as:

```
Intent intent = new Intent( Action, Data );
```

such that Action is a Java like package name (e.g., fr.inrialpes.exmo.rdfoid.GETRDF) and the Data can be anything but would generally be a URI (e.g., content://contacts/people/22) and optionally a mime-type specifying the expected result. When called through:

```
startActivity( intent );
```

The targeted component can be explicit, i.e., identified the components to deal with the request, or Android can look for an application/component able to answer the Action on the Data and pass it the call and the arguments.

2.2 Android Content Providers

Android is built around different kinds of services, one of which being `ContentProvider` which exposes some data of an application to other applications. Content providers enable the transfer of structured data between device applications. They encapsulate the data and control the access to it through an interface. This interface empowers one to query the data or to modify it.

A content provider[2] is a subclass of `ContentProvider` and implements the following interface:

```
Cursor query( Uri id, String[] proj, String select, String[] selectArgs, String orderBy )
Uri insert( Uri id, ContentValues colValueList)
int update( Uri id, ContentValues colValueList, String select, String[] selectArgs )
int delete( Uri id, String select, String[] selectArgs )
String getType( Uri id ) .
```

which allows one to query, to insert, to delete or to update the data . Queries are issued in an SQL manner and the results are presented to the user as a table.

[1] http://developer.android.com/guide/topics/fundamentals.html

[2] http://developer.android.com/guide/topics/providers/content-providers.html

2.3 Android URIs

With the content provider API, each data (table or individual) is identified by a URI having the following structure:

```
content://authority/path/to/data
```

The `content:` scheme is the cornerstone of each Content Provider URI, the `authority` identifies the provider, i.e., the dataset, and the `path/to/data` identifies a particular table or individual (row) in the dataset. For example, the URI `content://contacts/people` refers to all the people in the contact application, and the URI `content://contacts/people/33` identifies a specific instance of these, namely the instance having the id 33.

When an application wants to access a particular piece of data, it queries its URI. This is done through a call to the `ContentResolver` which is able to route the query to the right content provider.

From a semantic web point of view, using URIs to identify data is a strong point of the Android content providers. Still, there are several limitations if we would like to use them as a linked data interface.

Specifically, URIs used by content providers are local to each device, i.e., not dereferenceable on the web, and not unique. The content scheme used by providers is not a standard protocol. Furthermore, two distinct devices will use the same URI to identify different data. For example, by using `content://contacts/people` one would be able to access the contacts from both devices.

3. The RDF Content Provider Framework

We designed the `RDFContentProvider` framework to offer uniform access to phone data and to give a semantic web flavour to Android. It is composed of the `RDFContentProvider` API and the `RDFContentResolver` application. The API must be included inside the applications that want to access RDF providers and inside the applications that want to define new RDF content providers. The `RDFContentResolver` application is the one that records all the RDF content providers installed on the device and routes queries to the relevant provider. Figure 3.1 gives an overview of the framework architecture.

3.1 The RDF Content Provider API

The goal of the `RDFContentProvider` API is to answer to two types of queries:

- Queries that request information about a particular individual (i.e., tell me what you know about individual X). The provided answer would be made of a set of triples which will correspond to the description of one object and the attribute values.
- Queries that request only the values for some variables that must satisfy a specific condition (i.e., SPARQL-like queries). In this case, the answer should correspond to a table of tuples, like in `ContentProviders` or SPARQL.

The `RDFContentProvider` API delivers the following classes and interfaces:

- `RdfContentProvider`: An abstract class that should be extended if one wants to create an RDF content provider. It subclasses the `ContentProvider` class belonging to the Android framework;
- `RdfContentResolverProxy`: A proxy used by applications to send queries to the `RDFContentResolver` application;
- `Statement`: A class used for representing an RDF statement;
- `RdfCursor`: An iterator on a set of RDF statements;
- `RdfContentProviderWrapper`: A subclass of `RdfContentProvider` which allows for adding RDF content provider capabilities to an existing classical content provider.

`RDFContentProvider` follows primarily the same kind of interface as `ContentProvider`. The minimal interface to implement linked data applications is:

- `RDFCursor getRdf( Uri id )`

The Cursor iterates on a table of subject-predicate-object (or predicate-object) which are the triples involving the object given as a URI. If one wants to offer a more elaborate semantic web interface, i.e., a minimal SPARQL endpoint, the following methods have to be also implemented:

- `Uri[] getTypes( Uri id )`: returns the RDF types of a local URI;
- `Uri[] getOntologies()`: ontologies used by the provider;
- `Uri[] getQueryEntities()`: classes and relation that the provider can deliver;
- `Cursor query( SparqlQuery query )`: returns results tuple;
- `Cursor getQueries()`: triple patterns that the provider can answer.

The RDF providers that we have developed so far are implementing only the first three primitives. This interface corresponds to the one we required to web services in our work on ambient intelligence [4]. Indeed, to some extent, this work is similar to the work in ambient

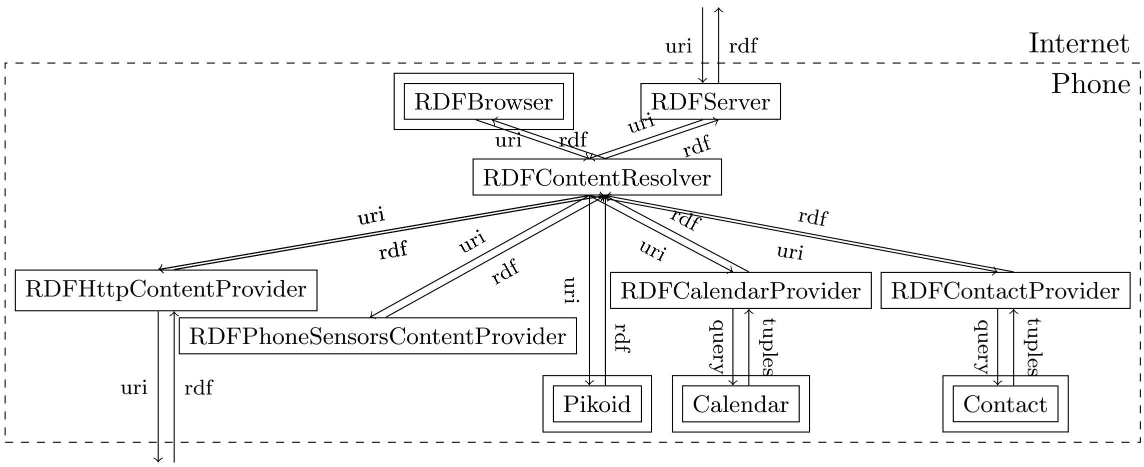


Figure 3.1: The architecture components and the communication between them. Components with double square have a graphic user interface.

intelligence except that instead of dealing with a building-like environment, it works within the palm of one's hand. But the problem is the same: applications which do not know each others and have to communicate through semantic web technologies.

3.2 The RDF Content Resolver Service

The `RDFContentResolver` service has the same goal as the `ContentResolver` belonging to the Android framework. It maintains the list of all the installed RDF content providers, and forwards the queries it receives to the corresponding one. This application is never visible to the user, therefore we have implemented it as an Android service.

When an RDF Content Provider is instantiated by the system, this provider automatically registers to the `RDFContentResolver`. A principle similar to the one from the Android Content Provider framework is used.

The `RDFContentResolver` can route both the local (`content:`) and external (`http:`) URI-based queries. In case of a local URI, i.e., starting with the `content` scheme, the resolver decides to which provider it must redirect the query. In case of an external URI, i.e., starting with the `http` scheme, the provider automatically routes the query to the `RDFHttpContentProvider` (see Figure 3.1).

The `RDFHttpContentProvider` allows one to retrieve RDF data from the Web. It parses RDF documents and presents them as `RDFCursors`. So far, only the minimal interface has been implemented, i.e., the `getRdf( Uri id )` method.

3.3 RDF Provider wrappers for Phone applications

The RDF Content Resolver application is also bundled with several RDF content providers encapsulating the access to Android predefined providers. The Android framework has applications that can manage the address book and the agenda. These two applications store their data inside their own content provider.

In order to expose this data as RDF, we developed the `RDFContactProvider` and the `RDFCalendarProvider`. These providers are wrapper classes for the ContactProvider and the

CalendarProvider residing inside the Android framework. The same could be obtained by bypassing their ContentProvider interface and using instead the W3C Device APIs since they exist for each of these applications[1]. So doing should not be significantly more difficult and would ease porting to other platforms than Android.

`RDFContactProvider` exposes contact data using the FOAF ontology. It provides data about a person's name (display name, given name, family name), about his phone number, email address, instant messenger identifiers, homepage and notes.

`RDFCalendarProvider` provides access to the Android calendar using the RDF Calendar ontology[2]. The data supplied by this provider contains information about events, their location, their date (starting date, ending date, duration, and event time zone), the organizer of the event and a short description.

`RDFPhoneSensorsContentProvider` aims to expose sensor data from the sensors embedded inside the mobile device. Contrary to the others, they are not offered as Content Providers. At the present time, it only delivers the geographical position (retrieved using the Android LocationManager service). In order to express this information in RDF, we use the geo location vocabulary[3], the one that provides a namespace for representing lat(itude) and long(itude).

Developing a wrapper would consist, in general, of the following steps:

– Identify data exposed in the application content provider;
– Choose ontologies corresponding to this data;
– Provide a URI pattern for each ontology concept;
– Implement a dereferencing mechanism which, for each type of resource, extract information from the content provider and generates RDF from this (generating URIs for related resources).

A native RDF Content Provider application may follow the same steps. However, it may be developed without any Content Provider. In this case, the analysis has to be carried out from the application data (or a corresponding API).

3.4 Pikoid

Pikoid is a simple application allowing a user to annotate on the phone pictures with people from the address book, places from the map and events from the calendar. It uses the foaf and vcal ontologies. It natively implements the RDF Content Provider interface. Figure A.1 shows browsing starting from Pikoid.

[1] `http://www.w3.org/2009/dap/` or `http://www.w3.org/TR/geolocation-API/`
[2] RDF Calendar vocabulary: `http://www.w3.org/TR/rdfcal/`.
[3] Geo location vocabulary: `http://www.w3.org/2003/01/geo/` .

4. RDF Browser

The RDF Browser acts like a linked data client. Given a URI, the browser makes an HTTP URI request in order to retrieve the information from the specified location. It works both with `http:` and `content:` URIs. If the data contains other URIs, the user can click on them and the browser will issue a new query with this URI.

An example can be found in Figure 4.1. In this case, the user uses the `RDFBrowser` to get the information about the contact having the id 4. When the browser receives the request, it sends it further to the `RDFContentResolver`. Since the URI starts with the `content://` scheme and has the `com.android.contacts` authority, the resolver routes the query to the `RDFContactProvider`. This provider retrieves the set of triples describing the contact and sends it to the calling application which displays it to the user. Thereupon, the user decides that he wants to continue browsing and selects the contact's homepage. In this case, since the URI starts with the `http://` scheme, the resolver routes the query to the `RDFHttpContentProvider`. The same process repeats and the user can see the remote requested file, i.e., Tim Berners-Lee FOAF file.

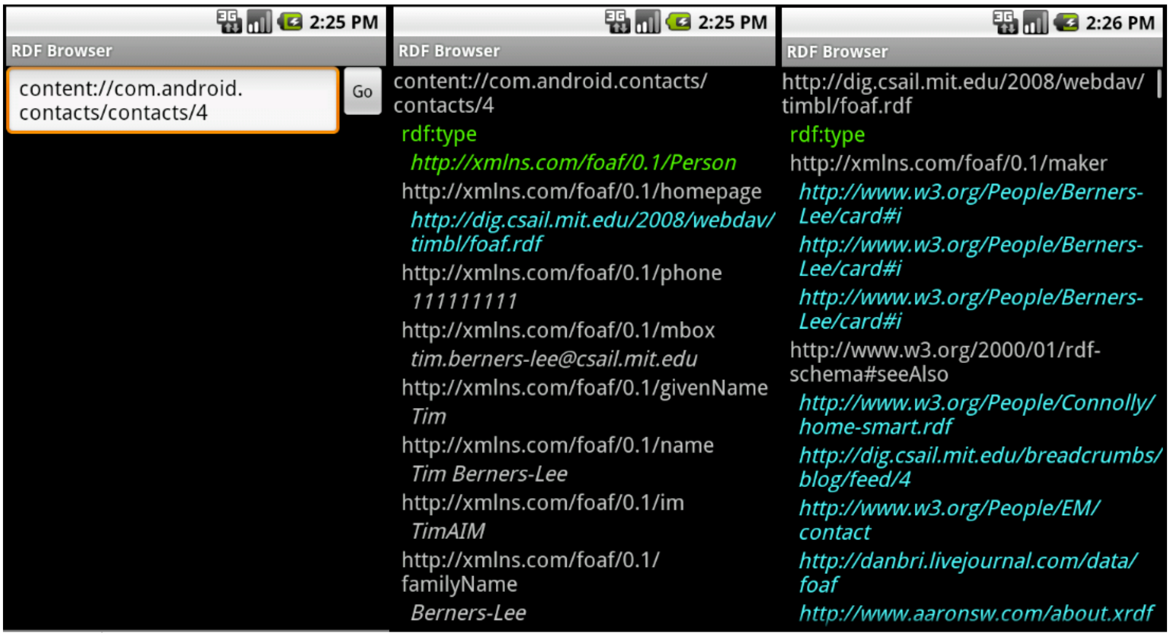


Figure 4.1: An example of using the RDF Browser.

5. RDF Server

The RDF Server is a new component added to the architecture. This server provides to the outside world the data stored into the device as RDF. Due to the fact that the server must maintain a permanent connection to the Internet without user interaction, we implemented it as an Android service, i.e., a background process.

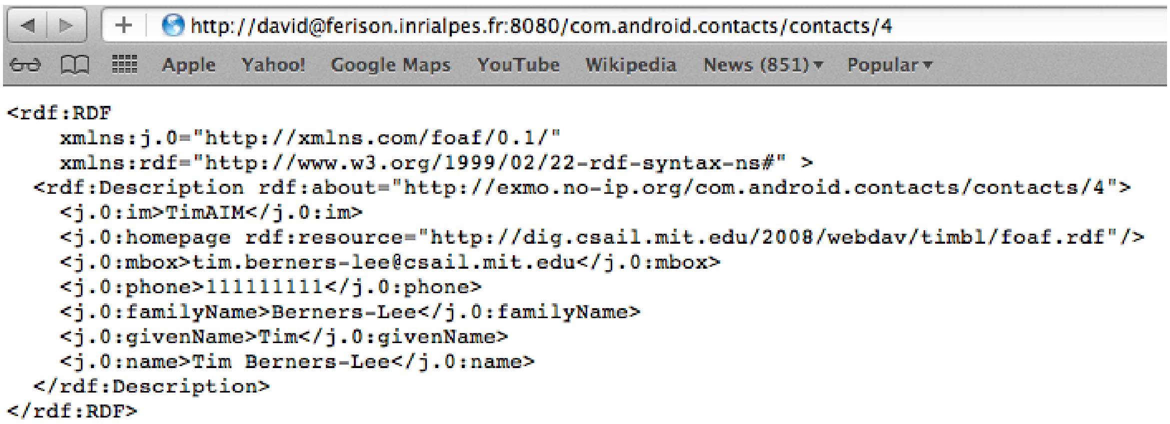
One important issue appears when one would like to get data from a device because the URI used to query the content providers has a local meaning. In the outside world, the URI used to query the address book of two different people will be the same, but the content of the address book will be different.

The server principles are quite simple. In the beginning, the server receives a request from the outside. Then, it dereferences the requested URI, i.e., it translates the external URI into an internal one, which has meaning inside the Android platform. The RDF Server sends it further to the `RDFContentResolver`. In a manner similar to the one explained for the RDF Browser the set of triples is obtained but, before sending this set to the server, the URIs of the triples are externalized and the graph is serialized using a port of Jena under the Android platform.

The URI externalization process translates the local URI `content://authority/path/to/data` into the dereferenceable one `http://deviceIPAddress:port/authority/path/to/data`. Reversing the translation of such a URI is possible since both the authority and the path are kept during the externalization process.

Usually, mobile devices do not have a permanent IP address and thus, the externalized URIs are not stable. To overcome this, a dynamic DNS client[1,2] can be used.

In addition, the server supports a minimal content negotiation mechanism. If one wants to receive the data in RDF/XML, it will set the MIME types of the Accept-type header of its request to “application/rdf+xml” or to “application/*”. In the opposite case or when the client sets the MIME type to “text/plain”, the data will be transmitted in the n-triple format. Not only the requester has the opportunity to express its preferences regarding the format of the received data, but the default format of the transmitted data can be specified in the server settings, as well the port on which the server can listen on and the domain name server for it.




```
<rdf:RDF
    xmlns:j.0="http://xmlns.com/foaf/0.1/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <rdf:Description rdf:about="http://exmo.no-ip.org/com.android.contacts/contacts/4">
    <j.0:im>TimAIM</j.0:im>
    <j.0:homepage rdf:resource="http://dig.csail.mit.edu/2008/webdav/timbl/foaf.rdf"/>
    <j.0:mbox>tim.berners-lee@csail.mit.edu</j.0:mbox>
    <j.0:phone>111111111</j.0:phone>
    <j.0:familyName>Berners-Lee</j.0:familyName>
    <j.0:givenName>Tim</j.0:givenName>
    <j.0:name>Tim Berners-Lee</j.0:name>
  </rdf:Description>
</rdf:RDF>
```

Figure 5.1: RDF Server response.

[1]Dynamic DNS Client: `https://market.android.com/details?id=org.l6n.dyndns&hl=en`.
[2]DynDNS: `http://dyn.com/dns/` .

An example can be found in Figure 5.1. In this scenario, the user retrieves information about the fourth contact from the device address book. The request is processed by the RDF Server in a manner similar to the one of the RDF Browser.

6. Technical issues: application size

The RDF Server included in our architecture eases the access of the user to the RDF data found on the web. For that purpose, we wanted to reuse an existing semantic web framework, such as Jena or Sesame. Yet they are not suitable to be employed under the Android platform (the code depends on some libraries that are unavailable under Android). There are a few ports of these frameworks to Android: Microjena[1] and Androjena[2] are ports of Jena and there exists a port of Sesame to the Android platform mentioned in [1]. We use Androjena.

A problem that arises when we use this framework is that the size of the application increases substantially. This problem could have been avoided by reimplementing only the Jena modules that are needed in our architecture. Still, we would like to improve our architecture by adding more features (such as a SPARQL query engine) that require additional modules to those used to read/parse/write RDF, available in Jena.

A tool that we found useful in our development process was ProGuard. ProGuard[3] is a code shrinker, optimizer, and obfuscator. It removes the unused classes, methods or variables, performs some byte-code optimizations and obfuscates the code. The tool proved to be efficient in reducing the size of our application (our framework including Androjena) by half, i.e., its initial size was 6.48MB, and after we applied the tool it diminished up to 2.98MB.

The existence of such tools as ProGuard, is a step forward in the continuous battle between applications that require a considerable amount of space for storing their code and devices with a reduced memory storage.

[1] `http://poseidon.ws.dei.polimi.it/ca/?page_id=59`.
[2] `http://code.google.com/p/androjena/`.
[3] `http://proguard.sourceforge.net/`.

7. Perspectives: SPARQL querying

We are currently examining how to query the device data using SPARQL. There are three main ways of doing this:

- creating a new RDF content provider which relies on a triple store to deposit the data [8], and then using SPARQL to query it; or
- translating SPARQL queries into SQL queries, and further decompose it in a form compatible with the ContentProvider interface.
- translating SPARQL queries into more specific request that may be answered by an RDF Content Provider (does not require to have a Content Provider or a SQL database).

Concerning the second option, there are several available tools that can make the translation from SPARQL to SQL, like Virtuoso or D2RQ. However, these tools solve only half of the problem because the SQL queries have to be adapted to the ContentProvider interface, i.e., the queries have a particular format, different than the SQL one. This interface allows for querying only one view of a specified table at a time, hence it is not possible to ask Content Providers to perform joins.

Further challenges regarding security must be taken into account. The user of the application should be able to grant or to deny the access to its personal data. A specific vocabulary should be used in order to achieve this [1]. Moreover, the dangers of granting system access to a third-party user can be avoided by using a secure authentication protocol [2].

Finally, the problem of resource consumption is mentioned here for the record. Such resource may be related to bandwidth (WiFi or 3G) that are consumed by having the RDF-Server working. In addition, such a server, and the use of our framework in general, may affect energy consumption. This will have to be precisely considered.

As can be seen, there are still technical problems in implementing a full RDF framework at the core of Android. Specific solutions must be developed.

[1] `http://www.w3.org/wiki/WebAccessControl`.
[2] `http://www.w3.org/wiki/Foaf+ssl`.

8. Conclusion

Involving Android devices in the semantic web, both as consumers and providers of data, is an interesting challenge. As mentioned, it faces the issues of size of applications and URI dereferencing in mobility situations.

The application described here are available from `http://swip.inrialpes.fr`.

A next step is to provide a more fine grained and structured access to data through SPARQL querying. Another extension when querying is the use of an alignment server in order to transform queries on the fly when different ontologies are used in the query and a corresponding content provider.

This promises to raise the issue of computation, and thus energy, cost on mobile platform.

A further issue will be the control of privacy in such a framework. But here too, we think that semantic technologies can help.

References

[1] Mathieu d'Aquin, Andriy Nikolov, and Enrico Motta. Building SPARQL-enabled applications with android devices. In *Proc. 10th ISWC demonstration track, Bonn (DE)*, 2011.

[2] Jérôme David and Jérôme Euzenat. Linked data from your pocket: The Android RDF-ContentProvider. In *Proc. 9th ISWC demonstration track, Shanghai (CN)*, pages 129–132, 2010.

[3] Jérôme David, Jérôme Euzenat, and Maria Rosoiu. Linked data from your pocket. In Christophe Guéret, Stefan Schlobach, and Florent Pigout, editors, *Proc. 1st ESWC workshop on downscaling the semantic web, Hersounissos (GR)*, pages 6–13, 2012.

[4] Jérôme Euzenat, Jérôme Pierson, and Fano Ramparani. Dynamic context management for pervasive applications. *Knowledge engineering review*, 23(1):21–49, 2008.

[5] Marko Gargenta. *Learning Android.* O'Reilly Media, Inc., 2011.

[6] Ora Lassila. Semantic web approach to personal information management on mobile devices. In *Proc. IEEE International Conference on Semantic Computing (ICSC), Santa Clara (CA)*, pages 601–607, 2008.

[7] Reto Meier. *Professional Android 2 Application Development.* Wrox, 2011.

[8] Danh Le Phuoc, Josiane Xavier Parreira, Vinny Reynolds, and Manfred Hauswirth. RDF On the Go: An RDF storage and query processor for mobile devices. In *Proc. 9th ISWC demonstration track, Shanghai (CN))*, November 2010.

A. Demo path

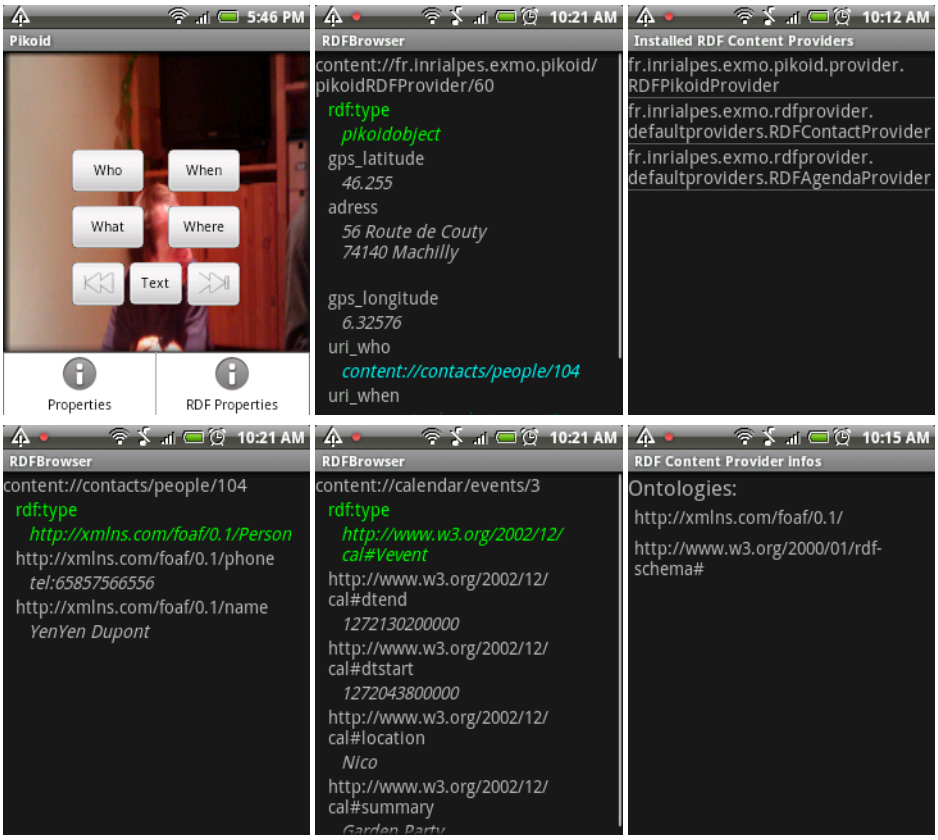


Figure A.1: The Pikoid application annotates images with metadata stored as RDF. RDFBrowser allows for querying this information to the Pikoid RDFContentProvider interface and displaying it. The current picture metadata is shown in the second panel (pikoidRDFprovider/60). From there, it is possible to browse the information available in the address book (people/104) and the calendar (events/3) through the corresponding RDF Content Providers wrapping them (see Figure A.2). Finally, RDFBrowser also allows for inspecting available RDFContentProviders and the ontologies they manipulate.

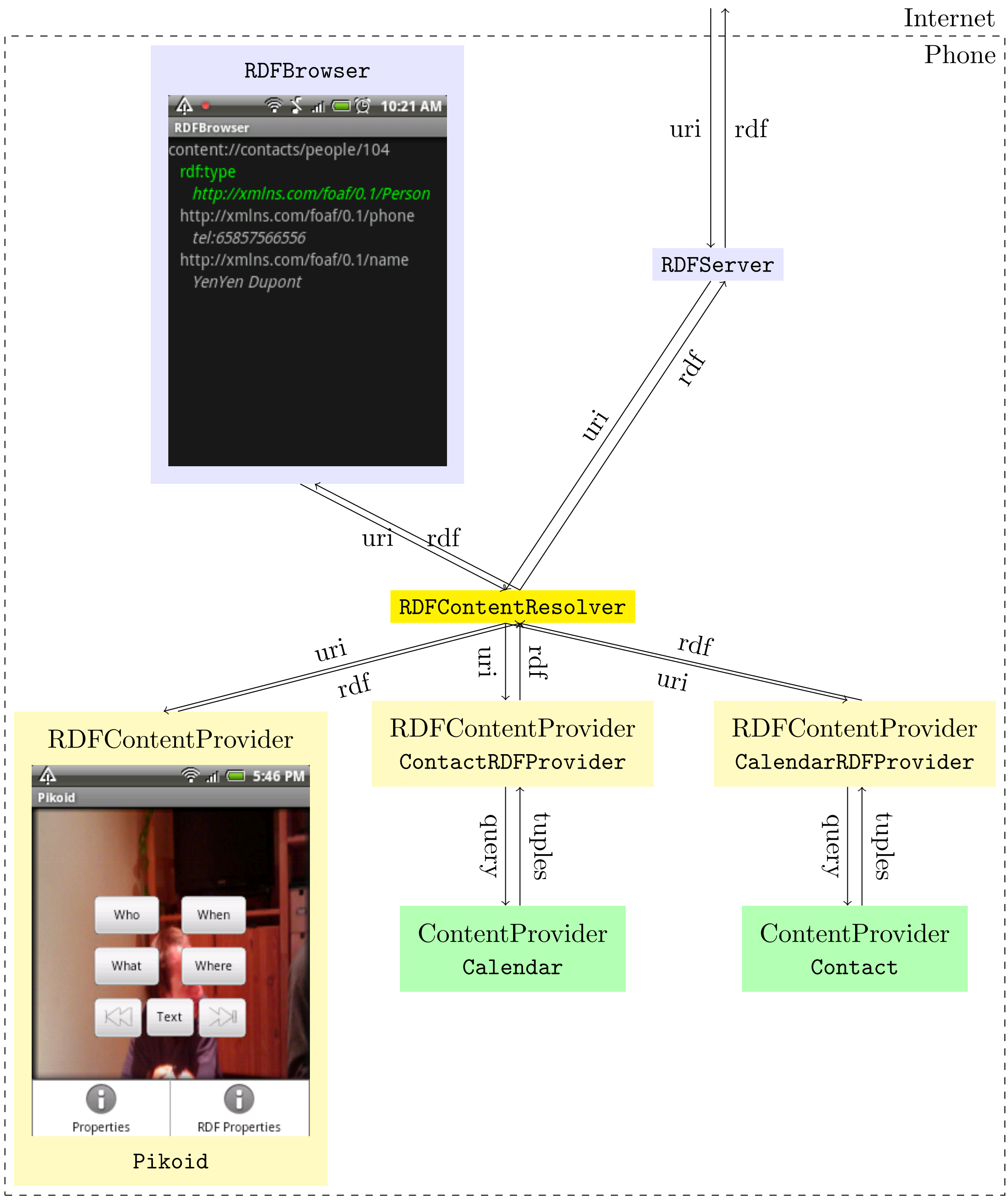


Figure A.2: Various components as implemented in the platform.