API Blender: A Uniform Interface to Social Platform APIs

Georges Gouriten
Institut Télécom; Télécom Paristech; CNRS LTCI
46, rue Barrault, Paris, France
georges.gouriten@telecom-paristech.fr

Pierre Senellart
Institut Télécom; Télécom Paristech; CNRS LTCI
46, rue Barrault, Paris, France
pierre.senellart@telecom-paristech.fr

ABSTRACT

With the growing success of the social Web, most Web developers have to interact with at least one social Web platform, which implies studying the related API specifications. These are often only informally described, may contain errors, lack harmonization, and generally speaking make the developer's work difficult. Most attempts to solve this problem, proposing formal description languages for Web service APIs, have had limited success outside of B2B applications; we believe it is due to their top-down nature. In addition, a programmer dealing with one or several of these APIs has to deal with a number of related tasks such as data integration, requests chaining, or policy management, that are cumbersome to implement. Inspired by the SPORE project, we present API Blender, an open-source solution to describe, interact with, and integrate the most common social Web APIs. In this perspective, we first introduce two new lightweight description formats for requests and services and demonstrate their relevance with respect to current platform APIs. We present our Python implementation of API Blender and its features regarding authentication, policy management and multi-platform data integration.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; D.3.2 [**Programming Languages**]: Language Classifications—*Python*

General Terms

Design, Standardization

Keywords

social Web, API, REST, data integration

1. INTRODUCTION

Interacting with platforms like Facebook, Youtube, Twitter, Flickr, or Google+ becomes an important part of many software projects, whether it is for authentication purposes, to collect information about a user, to present mash-ups of popular social Web data, or for a myriad of other reasons. Our perspective comes from the need of *archiving* important social data for preservation purposes.[1] Regular Web archives, such as those built by the Internet Archive[2], often include content from or pointers to social Web platforms but do not benefit from API data. As a consequence, the archives are either partial – Facebook disallows generic crawling of its public pages – or lack some extra information that the API can provide, for instance extracted entities on Twitter. Designing an *archival crawler* for the social Web requires interfacing with the multiple social Web APIs, as well as respecting the *policies* imposed by these services, such as limiting the number of requests per hour.

Many projects thus involve numerous interactions with various social platforms, sometimes with complex logics such as getting the social graph till the third rank of users having mentioned a specific keyword. Understanding the related API specifications can be challenging. There is no *de facto* standard to describe them and they can contain mistakes or approximations. There is no clear specification, for instance, of how many requests per hour are allowed on the Twitter search API. For the most popular platforms, specific language libraries sometimes exist but they often require the same learning phase.

Having a unified description of the different social Web APIs is a technical challenge. An early step was taken with WSDL [2], a Web Services Description Language standardized by the W3C. WSDL has been heavily used in the industry and is at the core of many service-oriented software projects [5]. However, most popular social platforms including Facebook, Twitter, or Google+ and many other Web services are not currently offering any WSDL description of their API and do not seem to have any plans to do so. The reasons are manifold: WSDL-based services are often considered heavy machinery for such lightweight interfaces [1], WSDL has historically focused on SOAP message exchanges rather than on RESTful APIs though it can now express both [10], WSDL has no support for important API metadata such as policy management or the description of a sequence of service calls[3]. In reaction to WSDL, some other approaches to Web services description have been proposed, a prime

[1]ARCOMEM project, `http://www.arcomem.eu/`
[2]`http://www.archive.org/`
[3]BPEL [8] is typically used in B2B projects that need service orchestration, but leads to even heavier machinery.

WWW 2012 Developer Track, April 18–20, 2012, Lyon, France.

example being WADL [6] but they have not met with more success on popular social Web platforms.

Another perspective is necessary. Spring Social[4] is a Java framework to interact with the different social platforms. We believe this bottom-up approach is a very promising way to make the developers' work simpler. Spring Social implements a number of useful functionalities (authentication, uniform interface to some of the API types, etc.) but does not fulfill our requirements. On the one hand, some important features, especially for archival crawling, are not considered, such as limits on the number of requests. On the other hand, using Spring Social requires understanding an important amount of code before being able to interact with a social platform. To give an order of magnitude of the size of the software, the core v1.02 contains 405 files, without implementing any Web API.[5] With API Blender, we aim at more simplicity and flexibility, as highlighted by the example of use we give in Section 3.

Our main source of inspiration has been the SPORE project [4]. It consists in a simple implementation-agnostic JSON format allowing to describe Web APIs designed according to the REST principles. The project has been started recently and is still under development.

With API Blender, we extend SPORE with the following contributions:

1. two simple description formats at the API and request levels, adapted to social platforms, sorting SPORE out and complementing it;
2. an open Python implementation, allowing to easily integrate various platforms;
3. the following features, some of them left out of existing tools or libraries: authentication, server policy management, multi-platform data integration, and request chaining.

We designed API Blender inspired by what we observed on five prominent social platforms we identified: Twitter, Facebook, Google+, FlickR and Youtube. However, we strove at keeping a high flexibility so that it can be extended to many other Web APIs.

Our article is organized as follows. In Section 2, we present descriptions formats and discuss their relevance to social platforms. We then detail in Section 3 our implementation in Python and its features.

2. DESCRIPTION FORMATS

A Web API consists in a set of HTTP request messages associated to responses, sent to a specific HTTP server having its own rules. Note that Twitter has different APIs corresponding to different hosts: for instance, `api.twitter.com:80` or `search.twitter.com:80`.[6] We describe a Web API with several objects that allow to describe the server and its rules (with respect to access policies) as well as the interactions it offers. We find JSON [3] light and readable and have chosen to use it as a serialization. In what follows, we tried using straightforward names and self-explaining conventions to define the different elements.

[4] `http://www.springsource.org/spring-social`
[5] `http://s3.amazonaws.com/dist.springframework.org/release/SOCIAL/spring-social-1.0.2.RELEASE.zip`
[6] `https://dev.twitter.com/docs/history-rest-search-api`

Server description format. We have extended SPORE with a consistent oriented-object approach, as well as the addition of authentication and policy usually required to interact with social platform Web APIs.

Server Object

```
"name": string,
"host": string,
"port": integer,
"authentication": auth_object,
"policy": policy_object,
"interactions": [interaction_object]
```

Port, policy, and authentication are optional. The port defaults to 80.

Two authentication protocols are supported at the moment, one based on a unique authentication URL with parameters and the other on the three-legged OAuth2 [7].

Simple Authentication Object

```
"request_token_url": uri,
"url_parameters": object
```

By simple authentication, we mean authentication with parameters such as API key or login and password passed to a unique URL so as to receive the authentication token.

OAuth2 Authentication Object

```
"consumer_key": string,
"consumer_secret": string,
"request_token_url": uri,
"access_token_url": uri,
"authorize_url": uri
```

Many social platforms (e.g., Twitter, Facebook, Google+) accept OAuth2 authentication.

Policy Object

```
"requests_per_hour": integer,
"too_many_calls_response_code": integer,
"too_many_calls_waiting_seconds": integer
```

An overload can be detected by counting the requests or receiving a too-many-calls response. In the latter case, API Blender will snooze for the specified amount of time before testing if the counter has been reset.

Interaction description format. An interaction is a class of HTTP requests with a common root path and their associated responses. Here also we extended SPORE and added the response object.

Interaction Object

```
"name": string,
"description": string,
"request": request_object,
"response": response_object
```

The description is optional.

Request Object

```
"root_path": string,
"method": string
```

```
"raw_content": string
"url_parameters": [
  [ string, # key, e.g., "id"
    string, # type, e.g., "integer"
    boolean # is it an optional parameter?
    object  # the default value, it can be null ]
]
```

The method has to be GET, PUT, POST or DELETE. Providing raw content is optional and useful only for PUT and POST methods. If a default value is set on a URL parameter, it will be automatically passed with the default value unless it is explicitly set as null. This feature can be useful in many case such as requesting a default value of 100 responses per pages for a full-text query on Twitter search API.

Response Object

```
"expected_status_code": integer,
"serialization_format": serialization_format,
"expected_schema": json_schema_object,
"integration": extractor_object
```

The expected code is optional and defaults to 200. The serialization format has to be JSON or XML at the moment. The expected schema of the response is optional and can be defined as a JSON schema [9]. At the moment, we define a simple extractor that allows a mapping between a unified model and response fields. We use '.' as a path separator. For instance, we could have `"post.content": "post_data.text"` if our integrated model was `{"post": {"content": string}}` with a response model of `{"post_data": {"text": string}}`. With a careful normalization model (for instance using concepts of an ontology), this allows to integrate data coming from different platforms. As an extension, this semantic model could also be used to describe the inputs of services, a first step towards semantic service orchestration.

3. THE PYTHON IMPLEMENTATION

Python is becoming increasingly popular among developers. On the social coding platform GitHub, it is ranked third.[7] We find Python to be simple, flexible, and to have many useful libraries. We have chosen to implementation API Blender in this language. API Blender is available online at `https://github.com/netiru/apiblender`.

Structure. The module structure offered by Python allows us to adopt the following light structure.

API Blender package

```
main.py             Controller
server.py           Server and interactions
policy.py           Policy management
auth.py             Authentification management
config/             JSON configuration files
--general.json      General config
--apis/             API config files
```

We found it convenient to have one file per API server where we gather the descriptions for the server and its interactions. Currently, the API Blender supports the two Twitter APIs (generic and search), Facebook, Google+, FlickR and Youtube.

[7]After JavaScript and Ruby, `https://github.com/languages`

Features. API Blender implements several precious feature. It supports *the two main authentication types*: using a single URL with parameters and OAuth2 [7] thanks to Python OAuth2[8].

API Blender also *ensures respect of the server policy*; when the hourly limit is reached or when a too-many-calls response is identified, the policy manager will stop for some time and periodically test if the counter has been reset. *Error handling* is taken into consideration too, whether it regards a non-conforming configuration file or an unexpected response. Finally, API Blender gives the possibility to extract and normalize elements from responses. This feature supports simple field extraction and standardization at the moment but the same process will be possible with arbitrary subtree transformations in the near future.

Request chaining. The open nature of API Blender combined to the flexibility of Python can fill many needs. Request chaining becomes very simple with Python and complex interactions can become easy-to-maintain Python libraries. We illustrate this with the following example on two Twitter APIs. The program below retrieves the last three pages of tweets containing the keywords "good spirit" then fetches the local social network (followers and followees) of the authors of the tweets.

Example of request chaining with Python

```
import apiblender

blender = apiblender.Blender()

# Retrieving 3 pages of result
blender.load_server("twitter-search")
blender.load_interaction("search")
users = set()
for p in range(1,3):
    blender.set_parameters({"q": "good spirit",
                            "page": p})
    response = blender.blend()
    ts=response["prepared_content"]["results"]:

    for twitt in ts
        users.add(twitt["from_user"])

# Retrieving followers / followees for each user
blender.load_server("twitter-generic")
for user in users:
    blender.load_interaction("followers")
    blender.set_parameters({"screen_name":user})
    followers = blender.blend()

    blender.load_interaction("followees")
    blender.set_parameters({"screen_name":user})
    followees = blender.blend()

    # Printing everything
    print("User Name: %s" % user)
    print("\tFollowers: %s" % \
            followers["prepared_content"])
    print("\tFollowees: %s" % \
            followees["prepared_content"])
```

[8]Created and maintained by SimpleGeo Inc. `https://github.com/simplegeo/python-oauth2`

4. CONCLUSIONS

API Blender has been designed in the context of the ARCOMEM project on social Web archiving, and is put to use in this project to crawl and integrate data from various social Web platforms. We have found its flexibility useful in the light of the dynamicity of social Web platforms and managed to conveniently integrate the five platforms currently supported: Twitter, Facebook, FlickR, Google+, and Youtube. It is of potential use in any application that needs to access similar REST-inspired Web APIs and to export responses in a common schema. The source code being available on GitHub, we hope to solicit contributions, either in the form of extensions of the base functionalities, or in that of API descriptions. For future work, we see many promising opportunities such as:

1. smarter processing of responses, making use of the semantics of the services described, in the spirit of the semantic description of Web services à la OWL-S [11];
2. developing more standard request chaining libraries;
3. a possible integration of the different input schemas;
4. more research for a smarter snooze management;
5. distributing requests across different servers.

They all require to be very conscious of the existing trade-off between completeness and flexibility.

5. ACKNOWLEDGMENTS

The described work was funded by the European Union Seventh Framework Programme (FP7/2007–2013) under grant agreement 270239 (ARCOMEM).

6. REFERENCES

[1] G. Alonso. Myths around web services. *IEEE Data Eng. Bull.*, 25(4):3–9, 2002.

[2] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) 2.0. `http://www.w3.org/TR/wsdl20/`, June 2007. W3C Recommendation.

[3] D. Crockford. The application/json media type for JavaScript object notation (JSON). `http://www.ietf.org/rfc/rfc4627.txt?number=4627`, July 2006. IETF, Network Information Group.

[4] F. Cuny. SPORE – Specifications to a POrtable Rest Environment. `https://github.com/SPORE/specifications`, November 2011.

[5] T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[6] M. Hadley. Web Application Description Language. `http://www.w3.org/Submission/wadl/`, August 2009. W3C Member Submission.

[7] E. Hammer, D. Recordon, and D. Hardt. The OAuth 2.0 authorization protocol. `http://tools.ietf.org/html/draft-ietf-oauth-v2-23`, January 2012.

[8] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, April 2007.

[9] E. K. Zyp. A JSON media type for describing the structure and meaning of JSON documents. `http://tools.ietf.org/html/draft-zyp-json-schema-03`, November 2010.

[10] L. Mandel. Describe REST Web services with WSDL 2.0. `http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/`, May 2008. IBM Technical Library.

[11] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic Markup for Web Services. `http://www.w3.org/Submission/OWL-S/`, November 2004. W3C Member Submission.