Towards a Composition-Based APIaaS Layer

Claudio Guidi[1], Saverio Giallorenzo[1], Maurizio Gabbrielli[1]
[1]*Department of Computer Science and Engineering - DISI , University of Bologna, Italy*
cguidi@cs.unibo.it, sgiallor@cs.unibo.it, gabbri@cs.unibo.it

Keywords: Service-Oriented Computing, Cloud Computing, API, Composition, SaaS, PaaS, APIaaS

Abstract: Application Programming Interfaces (APIs) are a standard feature of any application that exposes its functionalities to external invokers. APIs can be composed thus obtaining new programs with new functionalities. However API composition easily becomes a frustrating and time-costly task that hinders API reuse. The issue derives from technology-dependent features of API composition such as the need of extensive documentation, protocol integration, security issues, etc.. In this paper we introduce the perspective of the API-as-a-Service (APIaaS) layer as tool to ease the development and deployment of applications based on API composition, abstracting communication protocols and message formats. We elicit the desirable features of such a layer and provide a proof-of-concept prototype implemented using a service-oriented language.

1 INTRODUCTION

Application Programming Interfaces (APIs) are a standard feature of any application that exposes its functionalities to external invokers. APIs can be composed to create new applications at client or server side. In the former case composition takes the form of a *mashup* (Liu et al., 2007), whereas in the latter case it is usually realized within a specific application. Depending on the desired result, developers choose for a server- or client-side composition, although there are some cases in which only a server-side application can comply with the requirements of the project, e.g., : (i) the developer needs to publish the composition as a new API; (ii) the composition has some strategic value, therefore the developer does not want to publish it as a readable script at client-side; (iii) APIs could use API keys and other identification data that must remain private and/or pass only through certified channels like VPNs or SSH; (iv) privacy policies and national agreements might enforce a known, certified, and geographically restricted route for data; (v) API contracts require the user to be part of a specific federated network. For these reasons and since our interest regards API composition and publication, in this work we focus on server-side API composition.

Composition of APIs implies the choice of a set of different technologies that span from the implementation language to the communication and serialisation protocols used by the application. The choice on these technologies it is very important since it strongly affects development, deployment, and maintenance of the project. In addition, it is not clear what long-term implications could derive from the choice of a specific set of technologies at design-time. In order to clarify how technology could affect a server-side API composition project, let us consider the example of Fig. 1 which reports the architectural view of a realistic application, dubbed PhotoBlog. PhotoBlog manages the publication of photos and their captions, translated in several languages, on various blogging platforms. PhotoBlog derives from the composition of three APIs: (i) *image*, for image handling (resize, cut, etc.) and storage; (ii) *translation*, for the automatic translation of captions; (iii) *blogging*, for publishing images and captions as posts. Moreover, PhotoBlog itself exposes its own APIs to client interfaces.

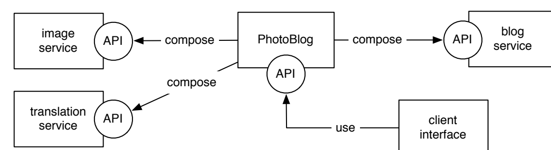

Figure 1: An example of composition of APIs.

Assuming T the technology chosen for the development of the project, the typical implementation of such a composition passes by the following phases:

- *documentation analysis*: the developer must understand the parameters of interaction of each API with T. Documentation analysis usually consists

of reading the companion documentation issued by the API provider;

- *API connection*: the developer must include, or develop by her own, specific libraries for T to handle communication and serialisation protocols with the composed APIs. Notably, each API could require the support of a different protocol;
- *development*: the developer implements the composition in T, employing the specific patterns of composition provided by T;
- *deployment*: the administrator of PhotoBlog deploys the application on a platform of a provider which supports T. Convenience, diffusion, and support for T is another strong concern which influences its choice;
- *maintenance and evolution*: the developer has to maintain the project up-to-date wrt new releases of APIs. API updates could require upgrades of T by either the administrator or the provider of the platform, depending on the service-level agreement.

Even in such a simple example as PhotoBlog, composing APIs requires a lot of "boilerplate" work to enable communication with the composed applications. This diverts the focus of the developer from the core composition functionality to the creation of a suitable environment for composition. In addition such an environment requires an exhausting work of maintenance.

What emerges from our brief analysis is that APIs are a commodity of any application, especially when deployed on the cloud, but the heterogeneity of protocols makes interacting with several applications an unaffordable task, if compared to its yield. The costs of setting a suitable environment easily overcome the potential value of the composition and represent an important drawback of programming web and cloud applications. This fact highlights the strong need for tools that enable efficient development of heterogeneous APIs, either in the cloud or on the Internet in general. However, to the best of our knowledge, currently there is no structured programming patterns nor formal theories which deal with API server-side composition in a cloud environment.

In this paper we try to attack this problem by introducing the notion of an API-as-a-Service (APIaaS) layer as a tool which eases the development and deployment of applications based on API compositions. By doing this, we also elicit the desirable general features of such a layer. Moreover, we also provide a proof-of-concept APIaaS prototype based on a service-oriented approach. In our vision, the APIaaS layer is a PaaS layer which provides all the tools for facilitating API composition, in particular by abstracting communication protocols and message formats. Service-oriented orchestration programming could be adopted as a programming reference for abstracting technological details and focusing on composition.

In § 2 we enhance the PhotoBlog example in order to list all the problems related on API composition. In § 3 we list all the features of an APIaaS layer should provide, whereas in § 4 we present our proof-of-concept APIaaS prototype which exploits a service-oriented approach both for the programming and the architectural level. Finally, in § 5 we discuss some related work, drawing our conclusions in § 6.

2 PROBLEMS WITH API COMPOSITION

In this section we list and discuss the main difficulties related to the development of applications that use and publish APIs. We extend the example of PhotoBlog, assuming T the technology selected for its implementation.

Documentation. Documentation regards the retrieval and integration of documents that describe APIs for the given technology T. In our example such a step must be performed for the three composed APIs, namely image, translation and blogging. Usually the documentation is described for RESTful JSON messages so, for each service, the developer has first to check the documentation in order to extract information on the identifier, the type, and the semantics of the values of request-response operations (if any). Then the documentation of specific libraries for T must be analysed, when supplied.

This process alone can take an important amount of time (Erl, 2004) when compared to that one of the development. Moreover, when the documentation is partial or missing the developer could end in a time-costly trial-and-error phase to acquire the information on how to interact with an API.

Interoperability. Interoperability concerns the mechanisms and technologies needed for requesting services and exchanging data between different APIs. In our example let us suppose that T natively supports interoperability with the translation and blogging APIs, but needs specific libraries for interacting with the image one. In this situation the developer faces two scenarios:

1. **Including libraries**. The developer integrates libraries of T which offer ready-to-use functionalities and reduce the workload of the developer.

However library inclusion causes the following drawbacks:

- strong *dependencies* in the project. Once included, dependencies need to be kept up-to-date and changes between versions may lead to major changes in the implementation of the applications;
- *conflicts* among libraries, source of bugs and unwanted side-effects;
- *limited knowledge* of the internal mechanisms of libraries. This constitutes a "grey area" of the project on which the developer has little to no knowledge, adding complexity to the debugging process.

2. **Ad-hoc solutions**. The developer could spend a significant amount of time in implementing *ad-hoc solutions* to support interoperability with the image API in T. In this case, she would need a comprehensive knowledge on the communication and serialisation protocols employed by the image API.

Publication. Published APIs allow interaction with client interfaces and other server-side applications. Publication concerns both APIs deployment and documentation, which represent two concerns:

- the problems addressed for interoperability hold also for API deployment. High levels of interoperability bring strong dependencies on several libraries, whilst supporting only a few protocols reduces the cost of implementation to the detriment of interoperability;
- publishing an API requires also the publication of its companion documentation. Such documentation is produced ad-hoc and requires a steady commitment as it must be up-to-date wrt the implementation it describes.

Security. Security regards secure transition, storage, and retrieval of data and it is a major issue of any application available on any open network. Implementing and administrating security requires specific knowledge, experience, and skills. Indeed the developer can easily include in T cryptographic libraries for secure communication like TSL and SSL and encryption/decryption libraries for data storage/retrieval. However security still requires a tremendous effort to ensure a specific level of reliability: a developer that chooses to implement and deploy the security of data on its own must be aware of the latest best practices (McGraw, 2004) in security. Moreover, she must invest a lot of effort to check the chain of components that handle sensible data to prevent security exploits (Barber, 2001).

In our example, PhotoBlog must provide APIs that enable users to set persistent configurations on the service. Users have to register to the service, link one or more blogs to their account, and identify themselves before publishing a post. This implies two APIs concerning security: registration/access of users and registration/access to blogs, depicted in Fig. 2.

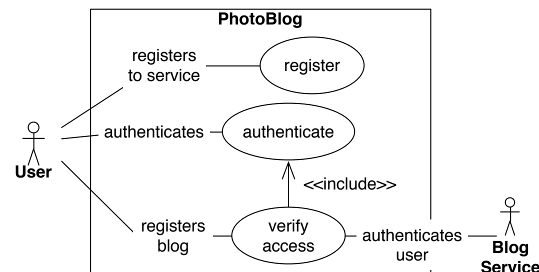

Figure 2: Users and blogs registration.

Monitoring. After PhotoBlog has been deployed, its administrator wants to check the performances of her application. An important step in project development is therefore *monitoring*. Monitorability (Skene et al., 2007) is a major concern when Service-Level Agreements (SLAs) define performances and availability of services with legal consequences such as penalties and compensations. As for security, T probably benefits from libraries to administer monitoring. In PhotoBlog the monitoring function, provided as an API, can serve both clients and its administrator. The administrator can monitor the activities on its application, checking for availability, performances, and resource requirements. Clients can also take advantage of the monitoring feature to assess the availability of the service. In addition, third parties can offer impartial monitoring of services to assess compliance with SLAs.

Portability. Portability is the degree of freedom of an application wrt its dependencies. *Internal* dependencies enforce the usage of libraries whose changes potentially lead to major rewrites of the application. *External* dependencies imply the presence of a specific elements in the running environment of the application. It could be the running environment itself, e.g., a specific interpreter, or other applications that must coexist in the same platform to enable a particular behaviour. In the previous paragraphs we extensively addressed the problems linked to internal dependency, however also external dependencies are a major issue which may lead to a *locked-in* situation. To illus-

trate this point let us say that the technology T chosen for PhotoBlog is Java, relying on Spring (Johnson et al., 2005) as supporting framework. Once released and deployed by the developer, the administrator of PhotoBlog is forced to choose a running environment that offers Java with Spring. In case the administrator wants to switch to another provider, she must choose one that offers a similar running environment. Otherwise she could spend a significant amount of time in setting the necessary running environment on a generic platform i.e., installing the JVM, the Spring framework, setting it properly, etc..

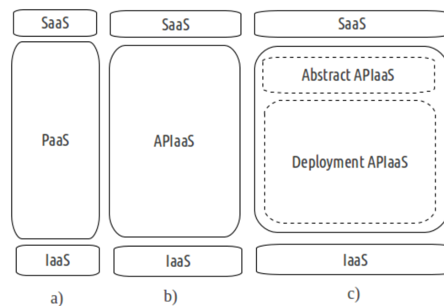

Figure 3: APIaaS layer in the common cloud stack.

3 FEATURES OF APIAAS

In this section we discuss the basic features which should characterize a composition-based APIaaS layer in a cloud framework. Notably, APIs can be divided into two groups, based on their implementation: *core* and *composed* APIs. Core APIs belong to applications that directly implement their functionalities without relying on external APIs (e.g., image, translation, and blogging APIs of PhotoBlog). Composed APIs derive from a behavioural composition of other APIs like the PhotBlog API. In our perspective an APIaaS layer should make no distinction in supporting core or composed APIs.

In this work we propose an APIaaS layer as a Platform-as-a-Service layer tailored to cope with the issues of API composition (3*b*). Developers can access such layer by means of private and secure credentials which give access to a private area that supports design of new API compositions starting from other existing core APIs. Developers can search existing APIs in the API registry of the APIaaS layer. APIs come with a formal interface which defines the message types and the behaviour of the interaction. They are also enriched with formal SLA documentation which provides all the necessary information for using/buying them. Each developer can decide to publish her APIs as *private*, *inner* or *public*. Private APIs are visible only by the author, inner APIs are visible only within the same APIaaS platform, whilst public APIs are visible from outside. Notably, inner and public visibility does not imply also accessibility and inner and public APIs can be released for free or under payment. Deployment costs and revenues depend on the business model of the APIaaS provider. Finally each deployed API should be monitorable and its performances visible by the administrator of the API and its clients, guaranteeing impartial SLA conformance.

Coherently with this scenario we propose to divide the APIaaS layer into two main sub-layers: the *Abstract APIAaaS* layer and the *Deployment APIaaS* one, as shown in Fig.3*c*. The former sub-layer supports the abstract composition of APIs. The latter provides all the necessary tools for their management. Such a division allows to separate programming issues, considered in the abstract layer, from the architectural ones, addressed by the deployment layer.

In the following, we list and discuss the main desirable features for these two sub-layers.

Abstract APIaaS

- *Composability*: is the availability of the linguistic tools that provide a suitable abstraction for achieving composition;
- *Formal documentability*: is the possibility of documenting and describing each API by means of formal and abstract interface specifications. Such specifications can be used to achieve composition abstracting the details of real communication protocols and message formats. Formal specifications enables the automatic extraction of stubs of descriptive documentation, thus helping developers in API documentation. In addition, semiautomatic production of documentation uniforms the overall documentation easing the interpretation of API documentation by clients. This feature is currently missing in RESTful APIs where there exist no formal nor standard language for defining API interfaces.
- *Portability*: deals with the possibility of moving an API composition from an APIaaS platform to another one without affecting its code. Portability is important to avoid lock-ins on APIaaS providers. Lock-in avoidance is one of the key to guarantee a competitive API market. Moreover, it also allows to separate functionality issues from the one related to deployment and performances.

Deployment APIaaS

- *Protection*: deals with the possibility of composing APIs within the APIaaS layer for inner usage, without publishing them thought a public Software-as-a-Service layer. This feature allows the creation of inner utility APIs used by more complex APIs.
- *Interoperability*: allows the APIaaS layer to connect with different applications and APIs by supporting several communication protocols and message formats (e.g., Http/JSON, Http/SOAP, Java RMI, XML RPC, etc.). This feature allows to connect the formal interfaces of the abstract layer to concrete communication protocols and interoperate with external APIs.
- *Publishability*: deals with the possibility to select and publish a subset of the developed APIs on the public SaaS layer. Such feature allows mashups and applications commonly developed with other technologies to employ the developed APIs.
- *Measurability*: concerns the possibility to measure the performances of APIs in terms of response time, memory usage, served sessions per time unit, etc.. All these information serve for the definition of SLA contracts both between APIaaS providers and API developers and between API developers and their clients. Since APIs can be internally composed, it should be possible to derive minimal SLAs starting from the composition of the SLAs of the composed APIs. As an example consider the case described in § 2. Let us suppose that the image blurring process takes at maximum 500msec whilst the process of image posting takes at maximum 200msec. Provided these data, a developer can guarantee in PhotoBlog's SLA that the response time of its image posting API has a upper bound of 700msec.
- *Sandboxing*: relates to the *protection* feature and extends it. In particular sandboxing of code enables each developer to access a different environment which runs and contains all her code. In the sandboxing environment the developer defines the visibility of its compositions i.e., which should be visible internally or which have public visibility for external usage.
- *Security*: deals with different aspects of security of the APIaaS framework, i.e., : *i*) *access security*; *ii*) *private data consistency*; *iii*) *malware prevention*. Access security deals with well-known security issues like authentication and access token release to allow clients access APIs; private data consistency deals with issues related to corruption of private data performed by malicious activities from the outside (e.g., hacking attacks that alter private data); malware prevention deals with issues related to malicious behaviour of the deployed composed APIs.
- *Scalability*: deals with the possibility to scale the APIaaS architecture depending on the number of subscribed developers and the number of developed APIs.
- *Federability*: deals with the possibility of federating different APIaaS layers in order to share the inner developed APIs among different providers. Such a feature allows for the creation of a horizontal layer among different cloud infrastructures and provides the abstraction of a unique and common development framework for APIs. Such federation could be useful for the creation of groups of specialised APIaaS providers which maximise variety, efficiency, and security of offered APIs. As an example, let us consider the case of a Smart City that providers the federation of three different APIaaS to citizens and companies: the municipality APIaaS, the main hospital APIaaS, and the public transportation company APIaaS. In this case, developers can easily access one of the three APIaaS and immediately exploit also all the APIs offered by the other two with the ease, efficiency, and security of the inner composition.

4 A COMPOSITION-BASED APIAAS PROTOTYPE

In this section we describe our composition-based APIaaS prototype, called JSOA, which is built following a service-oriented paradigm. The prototype represents an important enhancement wrt the PaaSSOA project (Guidi et al., 2012). In particular, JSOA introduces formal documentability, a web GUI designer, monitoring enhancements, publishability, and a prototype of the SaaS layer.

4.1 Abstract APIaaS

In the Abstract APIaaS layer we follow service orchestration principles to supply a composition language for APIs. In particular we use Jolie (Jolie, 2013) as orchestration language because it combines all the basic mechanisms of the well-known web services standards WSDL (W3C, 2001) and WS-BPEL (OASIS, 2007) in the same linguistic domain. In the following we discuss how composability, formal documentability, and portability are achieved by the JSOA Abstract APIaaS layer.

```
main {
setIntoBlog( r )( p ) {
  getImgWithFaceBlur@
    ImageAPI( r.image )( s.image )
  |
  getEngTranslation@
    TranslationAPI( r.text )( s.text );
  s.username = r.username;
  s.password = r.password;
  setPost@BlogAPI( s )( p )
  }
}
```

Listing 1: Composition behaviour of PhotoBlog 2 in Jolie.

Composability JSOA addresses composability in a natural way using the linguistic primitives supplied by an orchestration language (e.g., Jolie) which easily permit to coordinate concurrent service calls, thus focusing the value on composition and not on other aspects of programming. In particular these primitives cover the following aspects: *i*) *workflow behaviour*: the behaviour is expressed in terms of a workflow that provides both sequence and parallel operators, therefore no thread programming is required; *ii*) *synchronous invocation primitive*: APIs can be invoked by exploiting blocking request-response primitives which can be easily composed within the behaviour; *iii*) *statefulness primitives*: orchestration languages provide specific language primitives for addressing statefulness, e.g., such as correlation sets (Mauro et al., 2011) which ease session and state handling at application level; *iv*) *process spawn*: process spawn is automatically managed by the orchestration engines, no linguistic primitives are required; *v*) *endpoint abstraction*: endpoints allows to invoke APIs abstracting away from communication protocols and message formats. List. 1 reports a snippet of code corresponding to the behavioural composition of PhotoBlog in § 2. Notably, List. 1 does not report the declaration part which we excluded for reasons of presentation. In List. 1 we suppose that `getImgWithFaceBlur@ImageAPI` (Lines 3-4), `getEngTranslation@TranslationAPI` (Lines 6-7), and `setPost` (Line 10) are the names of the API for the image application, the translation application, and PhotoBlog respectively. Moreover, we define the name of the new composed API as `setIntoBlog` (Line 2). When the API is invoked (Line 2), the layer spawns an instance of its behaviour, executing it. In particular List. 1 composes the image and translation APIs in parallel (Line 5) and when they both return a response it commands the sequential execution of the subsequent instructions (Lines 8-9).

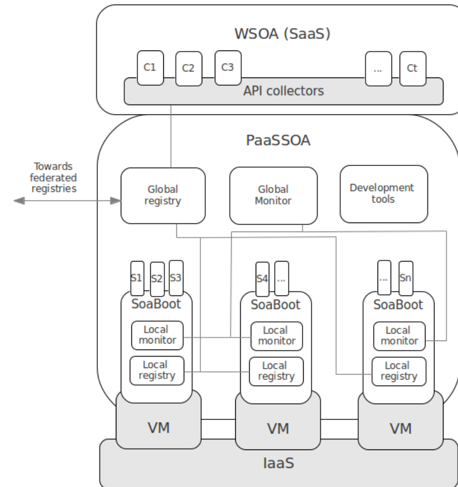

Figure 4: The JSOA Architecture.

Formal documentability In Jolie, each input and output endpoints of a service is joined with an interface which defines all the available APIs for the given port and the message types[1]. Each Jolie interface can be enriched with comments and automatically transformed into an human-readable HTML document.

Portability Portability differs if considered within the same technology or across different technologies. In the former case, JSOA Abstract APIaaS layer completely cope with portability issues. Since the Jolie interpreter is the same, compositions developed into the APIaaS can seamlessly execute without any significant modification of the code (except for endpoint rebinding which consists in a mere url location renaming) on any environment that provides said interpreter. On the contrary, in the latter case, portability is not directly achieved but since Jolie has been developed upon a formal semantics (Guidi et al., 2006) ad-hoc encoding translators can be developed.

4.2 Deployment APIaaS

Fig. 4 reports a sketch of the architecture of JSOA which implements the Deployment APIaaS layer. The main components are:

- *SOABoot*: the basic component which enables service execution in the cloud. It must be available in each computational resource provided by the IaaS and it takes the form of a service itself. SOABoot provides three main functionalities: (*i*) local

[1] Here we do not report the syntax of Jolie interfaces. The interested reader may refer to (Jolie, 2013)

service registration; (*ii*) local service monitor logs storage; (*iii*) local service execution.

- *PaaSSOA*: the component in charge of providing the PaaS abstraction to the developer. It equips a central registry that lists all running orchestrators in any SOABoot available. Registered orchestrators can be exploited for creating new service APIs. Moreover PaaSSOA manages a central monitoring service which collects all the logs from the running services.
- *WSOA*: the component that provides the SaaS abstraction and that is in charge of publishing service APIs outside JSOA.

At the present, JSOA does not respect all the features discussed in § 3. In the following, we briefly discuss the implemented ones whilst we address the missing ones in § 6 as future work.

Protection is inherently provided by JSOA since all deployed orchestrators and core APIs can be easily invoked by new API compositions within the Abstract APIaaS layer.

Interoperability is addressed by managing the deployment of the Jolie endpoints depending on the required protocols. Jolie provides both input and output endpoints used by an orchestrator to receive and send messages. As a test, we developed two core API wrappers that invoke WordReference (WordReference, 2013) and CKAN APIs (CKAN, 2013).

Publishability is implemented by the WSOA component which allows to select a set of already deployed APIs and to export them under different protocols (e.g., Http/JSON, Http/SOAP, Http/POST/XML, etc.). As a test, we developed a plugin that enables the composition of already published APIs within the editor of the SATIN project (Satin, 2013).

Measurability is provided by the monitoring architecture. A built-in monitor is attached to each deployed orchestrator. The monitor sends all the logged events to the SOABoot local monitor. In turn SOABoot local monitors send collected local data to the central monitor which stores them. The monitoring architecture also provides analysis of data via the Drools engine (Drools, 2013). The current implementation monitors start and end of sessions and start and end of operations. Although simple these data allows for the calculation of important statistics like API response time, memory usage, etc..

Scalability is addressed by SOABoot and the registry architecture. Depending on the load of the available computational resources new SOABoots can be added. In the same way, computational resources can be released when the corresponding SOABoots do not execute any orchestrator.

5 RELATED WORK

Recently the communities behind the main languages used for web development like JavaScript, Ruby, Java, and PHP produced several RESTful (Richardson and Ruby, 2007) libraries to ease the task of designing, writing, and composing RESTful APIs (some examples are, respectively, (Backbone.js, 2013), (Ruby on Rails, 2013), (Java Community Process, 2013), and (Lavarel, 2013)). RESTful frameworks ease the development of server-side API composition and deployment yet the implementation is part of the basic core of an application which translates into high costs of creation and maintenance.

An inspiring work in the direction of a PaaS middleware for application deployment is (Wettinger et al., 2013) where the authors address the problem of composition (and dependencies) with a middleware-oriented approach. In particular, we notice the focus on decoupling the application logic from the necessary middleware and infrastructure resources. On the same topic, other two interesting works are (Caporuscio et al., 2010; Caporuscio et al., 2014) where the authors introduce the concept of a middleware to leverage integration of heterogeneous services for ubiquitous networking. Such middleware is designed and implemented following a service-oriented approach supporting, respectively, SOAP and RESTful Services. This flexible and dynamic vision is also proposed in (Caporuscio et al., 2011) in which third-party applications discovered at run time should be integrated in applications to offer more sophisticated functionalities.

6 CONCLUSIONS

In this paper we discussed a new proposal for an composition-based APIaaS layer which facilitates API programming and deployment on the cloud by enabling API composition. In our perspective the APIaaS providers should supply large repositories of core APIs which developers could compose. Composed APIs could be available on the same repository of core APIs, enabling the publication of new

and more sophisticated services, thus fostering a liquid market for APIs and the reuse of services.

We also discuss a list of basic features which should be supported by an APIaaS layer and present a proof-of-concept prototype, dubbed JSOA, which implements a subset of such features. We based JSOA on a service-oriented paradigm and exploited primitives offered by its implementation language (Jolie) to orchestrate APIs and, more in general, to implement said features.

The aim of this work is to start a critical discussion towards a shared definition of the concepts and the foundational architecture of an open APIaaS layer.

Future work. We intend to continue the development of our prototype. Our experience with JSOA proved that prototyping constitutes a relevant resource for testing and analysing benefits and drawbacks of our approach. Namely, the next steps of JSOA development count the implementation of features like sandboxing, security, and federability.

In addition, we also plan to analyse how different economic models affect a federated market of APIaaS providers (as represented in the example regarding the *federability* feature in § 3). What we expect is that the features of our composition-based APIaaS layer would avoid technology lock-in, lower dependencies on specific technologies, and enact a beneficial competition among different providers.

REFERENCES

Backbone.js (2013). Backbone.js javascript framework. http://backbonejs.org/.

Barber, R. (2001). Hacking techniques: The tools that hackers use, and how they are evolving to become more sophisticated. *Computer Fraud & Security*, (3):9–12.

Caporuscio, M., Funaro, M., and Ghezzi, C. (2011). Restful service architectures for pervasive networking environments. In Wilde, E. and Pautasso, C., editors, *REST: From Research to Practice*, pages 401–422. Springer New York.

Caporuscio, M., Funaro, M., Ghezzi, C., and Issarny, V. (2014). *ubi*rest: A restful service-oriented middleware for ubiquitous networking. In *Advanced Web Services*, pages 475–500. Springer.

Caporuscio, M., Raverdy, P.-G., and Issarny, V. (2010). ubiSOAP: A Service Oriented Middleware for Ubiquitous Networking. *IEEE Transactions on Services Computing*.

CKAN (2013). Comprehensive knowledge archive network. http://ckan.org/.

Drools (2013). Drools - the business logic integration platform. http://www.jboss.org/drools/.

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

Guidi, C., Anedda, P., and Vardanega, T. (2012). Paassoa: An open paas architecture for service oriented applications. In *ESOCC 2012*, pages 208–209. Springer.

Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., and Zavattaro, G. (2006). Sock: A calculus for service oriented computing. In *ICSOC 2006*, pages 327–338.

Java Community Process (2013). The java api for restful web services. https://jcp.org/en/jsr/detail?id=339.

Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., and Kopylenko, D. (2005). *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK.

Jolie (2013). Java Orchestration Language Interpreter Engine. http://www.jolie-lang.org.

Lavarel (2013). The lavarel php framework. http://laravel.com/.

Liu, X., Hui, Y., Sun, W., and Liang, H. (2007). Towards service composition based on mashup. In *Services, 2007 IEEE Congress on*, pages 332–339. IEEE.

Mauro, J., Gabbrielli, M., Guidi, C., and Montesi, F. (2011). An efficient management of correlation sets with broadcast. In *COORDINATION*, pages 80–94. Springer.

McGraw, G. (2004). Software security. *Security Privacy, IEEE*, 2(2):80–83.

OASIS (2007). Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

Richardson, L. and Ruby, S. (2007). *Restful Web Services*. O'Reilly, first edition.

Ruby on Rails (2013). Ruby on rails. http://rubyonrails.org/.

Satin (2013). Satin project. http://www.satinproject.eu/.

Skene, J., Place, M., and Crampton, J. (2007). The monitorability of service-level agreements for application-service provision. In *WOSP '07*, pages 3–14. ACM Press.

W3C (2001). Web service definition language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

Wettinger, J., Andrikopoulos, V., Strauch, S., and Leymann, F. (2013). Enabling dynamic deployment of cloud applications using a modular and extensible paas environment. In *IEEE CLOUD*, pages 478–485. IEEE.

WordReference (2013). Wordreference. http://www.wordreference.com/.