2014 IEEE 6th International Conference on Cloud Computing Technology and Science

Leveraging Platform Basic Services in Cloud Application Platforms for the Development of Cloud Applications

Fotis Gonidis, Iraklis Paraskakis
South East European Research Centre
International Faculty of the University of Sheffield,
City College
Thessaloniki, Greece
{fgonidis,iparaskakis}@seerc.org

Anthony J. H. Simons
Department of Computer Science
The University of Sheffield
Sheffield, UK
a.j.simons@sheffield.ac.uk

Abstract—**Cloud application platforms gain popularity and have the potential to alter the way service based cloud applications are developed involving utilisation of platform basic services. A platform basic service is considered as a piece of software, which provides certain functionality and is usually offered via a web API, e.g. e-mail, payment, authentication service. However, the proliferation and diversification of platform basic services and the available providers increase the challenge for the application developers to integrate them and deal with the heterogeneous providers' web APIs. Therefore, a new approach of developing applications should be adopted in which developers leverage multiple platform basic services independently from the target application platforms. To this end, this paper presents a development framework whose objective is to enable the consistent integration of the platform services, and to allow the seamless use of the concrete providers by alleviating the heterogeneities among them.**

Keywords-Platform Basic Services; Cloud Application Platform; Service-based Cloud Applications; PaaS

I. INTRODUCTION

The proliferation of platform basic services has the potential to lead to a paradigm shift of software development where the former act as the building blocks for the creation of service-based cloud applications. Applications do not need to be developed from scratch but can rather be constructed using, where appropriate, various platform basic services, offered via the *cloud application platforms* [1].

However, these opportunities are accompanied by a number of challenges. The first challenge arises from the fact that there exist multitudes of a particular service, e.g., mail service, since the services are offered by many different providers. The second challenge arises from the need to provide a framework that spans across a number of different kind of services, i.e., mail services, billing services, message queue services and so on. The result of these two challenges implies that there exists a large heterogeneity among the offered services. The heterogeneity mainly arises due to (i) the differences in the workflow for the execution of the operations of the services, (ii) the differences in the exposed web APIs and (iii) the various required configuration settings and authentication tokens.

This paper proposes a development framework that tackles the three aforementioned challenges. The objective of the framework is two-fold: (i) First to enable the integration of platform basic services in a consistent way and (ii) second to facilitate the seamless use of the *platform basic service* providers by alleviating the heterogeneities among them. Thus application developers can focus on the design of the application without dealing with the peculiarities of each provider.

The paper is structured as follows: Section II presents related work. Section III describes the development framework and the process of supporting additional platform services and providers. Finally, Section IV concludes the paper and discusses future work.

II. RELATED WORK

Related work on the field can be grouped into three high-level categories [2]: Library based solutions, Middleware platforms, and Model-driven Engineering (MDE) techniques.

Library-based solutions such as jclouds [3] provide an abstraction layer for accessing specific cloud resources such as compute, storage and message queue. While, library-based approaches efficiently abstract those resources, they have a limited application scope which makes it difficult to reuse them for accommodating additional services. Middleware platforms, such as mOSAIC [4] and OpenTOSCA [5], enable the deployment and management of applications and have the capacity to exploit multiple cloud platform environments. However, they usually require a specific programming paradigm and may additionally impose a performance overhead. In MDE-based initiatives, cloud applications are designed in a platform independent manner and specific technologies are only infused in the models at the latest stage. MODAClouds [6] and PaaSage [7] are both FP7 initiatives aiming at cross-deployment of cloud applications.

The solutions listed in this Section focus mainly on the cross-deployment of application and traditional cloud resources such as compute, storage and message queue. However, they do not support additional platform services offered via web API such as payment, authentication and message queue service. In addition, the client adapters used to address the variability in the providers' APIs are hardcoded and thus not directly reconfigurable. On the contrary, the vision of the authors is, via the proposed framework, to facilitate the use of platform basic services from heterogeneous clouds in a seamless manner.

III. Development Framework

Fig. 1 depicts the high-level architecture of the framework. The process of adding a new service and provider to the framework can be divided into two parts: (i) The modelling of the platform service workflow and (ii) the description of the web API. For each part three phases are required: (1) the modelling of the abstract functionality of the platform service, (2) the implementation of the vendor specific functionality, and (3) the execution of the service.

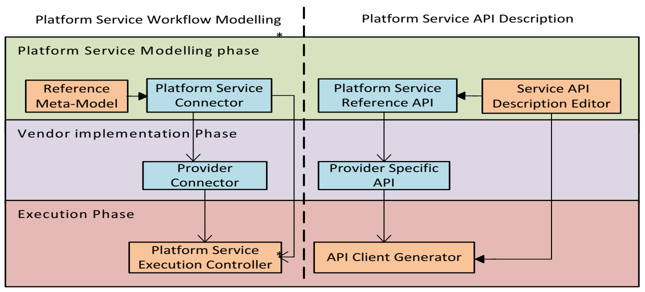

Platform Service Workflow Modelling
Platform Service API Description
Platform Service Modelling phase
Reference Meta-Model
Platform Service Connector
Platform Service Reference API
Service API Description Editor
Vendor implementation Phase
Provider Connector
Provider Specific API
Execution Phase
Platform Service Execution Controller
API Client Generator

Figure 1: High-level Overview of the Development Framework

In order to illustrate how the framework can be used in a real case scenario, the example of the cloud payment service is used in the rest of the paper. The payment service enables an application to accept online payments via electronic cards such as credit or debit cards.

Fig. 2 describes the steps involved in completing a payment transaction, while Fig. 3 shows the state chart of the cloud application throughout the transaction.

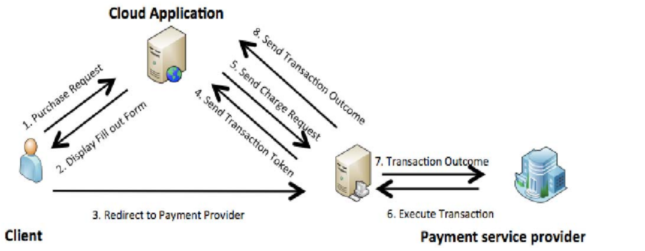


Figure 2. Cloud Payment Service

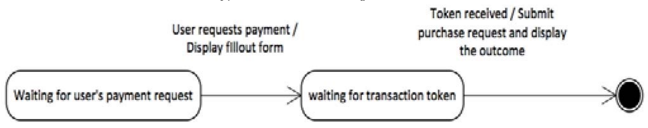

User requests payment / Display fillout form
Token received / Submit purchase request and display the outcome
Waiting for user's payment request
waiting for transaction token

Figure 3. State Chart of the Cloud Payment Service

Two states are observed. In the first state the cloud application waits for a purchase request and subsequently displays the fill out form. In the second state it waits for the transaction token issued by the payment provider and subsequently submits a charge request.

Next we describe in details the process of adding the payment service to the framework. As concrete payment provider, we use Spreedly, an add-on service provided by Heroku platform.

A. *Platform Service Workflow Modelling*

1) Platform Service Modelling Phase: During this phase, the abstract functionality of the platform service is modelled. For that reason the reference meta-model shown in Fig. 4 is used. The meta-model comprises the following components, which enable the modelling of the workflow during the execution of an operation:

CloudAction: CloudActions are used to model interaction with stateful platform basic services which require more than one step, to complete an operation. For each state, a separate CloudAction is defined to handle each incoming request and subsequently signals the transition to the next state.

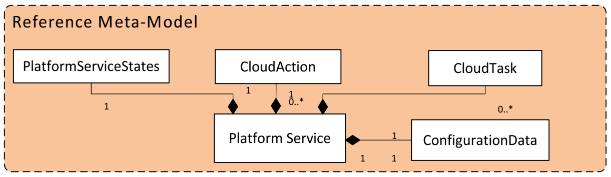


Figure 4. Reference Meta-model

*CloudMessage***.** CloudMessages can be used to perform synchronous requests to the service provider using the web API. The API usually conforms to the REST principles [8].

*PlatformServiceStates***.** The PlatformServiceStates file holds information about the states involved in an operation and the corresponding Cloud Actions which are initialised to execute the behaviour required in each state.

*ConfigurationData***.** This file contains certain configuration settings required by each platform service provider, e.g. client' credentials and authentication tokens.

The reference meta-model is used to construct the Platform Service Connector (PSC) as shown in Fig. 1. The PSC is a model of the abstract functionality of a given platform service. and is built based on the state chart defined for that service using the following rules: (i) For each state where the application waits for an incoming request, a CloudAction is defined to handle the request. (ii) For each outgoing request to the service provider using the web API, a CloudMessage is defined.

In the case of the Cloud Payment Service, the middle component of the Fig. 5 shows the Cloud Payment Service Connector. It is constructed based on the state chart defined in Fig. 3 and using the reference meta-model. It consists of the following blocks:

FilloutForm. It receives the request for a new purchase transaction and displays the fill out form.

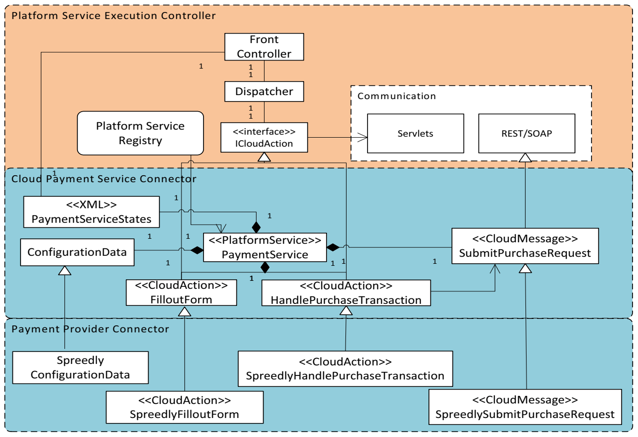
HandlePurchaseTransaction. It handles the transaction which involves receiving the transaction token, submitting the charge request and receiving and displaying the outcome.

SubmitPurchaseRequest. It is a CloudMessage used internally by the HandlePurchaseTransaction action. It submits a charge request to the service provider using the web API of the latter.

ConfigurationData. This file contains the settings required to complete the purchase operation and particularly the "redirectUrl", the username and the password.

PaymentSerivceStates. In the PaymentServiceStates file the states and the corresponding actions involved in the transaction are defined.

At this point the Cloud Payment Service Connector (PSC) does not contain any provider specific information. Therefore any payment service provider which adheres to the specified model can be accommodated by the abstract model.



Platform Service Execution Controller
Front Controller
Dispatcher
Communication
Platform Service Registry
<<interface>> ICloudAction
Servlets
REST/SOAP
Cloud Payment Service Connector
<<XML>> PaymentServiceStates
ConfigurationData
<<PlatformService>> PaymentService
<<CloudMessage>> SubmitPurchaseRequest
<<CloudAction>> FilloutForm
<<CloudAction>> HandlePurchaseTransaction
Payment Provider Connector
Spreedly ConfigurationData
<<CloudAction>> SpreedlyHandlePurchaseTransaction
<<CloudAction>> SpreedlyFilloutForm
<<CloudMessage>> SpreedlySubmitPurchaseRequest
1

Figure 5. Cloud Payment Service Model

2) Vendor Implementation Phase: After having defined the PSC, the specific implementation and settings of each concrete provider needs to be infused. For each CloudAction and CloudMessage defined in the PSC, the respective provider specific blocks should be defined forming the Provider Connector (PC).

The lower part of Fig. 5 shows the Cloud Payment Provider Connector for Spreedly. It contains the following blocks: (i) SpreedlyFilloutForm, (ii) Spreedly Handle Purchase Transaction and the (iii) Spreedly Submit Purchase Request. In addition, the ConifgurationData file needs to be updated accordingly in order to match the specific provider.

3) Execution Phase: During the execution phase the PSC and the PC, constructed in the previous phases, are managed by the Platform Service Execution Controller (PSEC) shown in the Fig. 5. The PSEC automates the execution of the workflow required to complete an operation. Its main modules are shown in the upper part of the Fig. 5.

Front Controller. The Front Controller serves as the entry point to the framework. It receives the incoming requests by the application user and the service provider.

Dispatcher. The dispatcher is responsible for receiving the incoming requests from the Front Controller and forwarding them to the appropriate CloudAction, through the ICloudAction. In order to do so, it reads the platform service states description file which contains the states and the corresponding actions.

ICloudAction. ICloudAction is the interface which is present at the framework at design time and which the Dispatcher has knowledge about. Every CloudAction implements the ICloudAction. That facilitates the initialisation of the new CloudActions during run-time.

Communication patterns. Two types of communication pattern are supported by the framework: Servlets which are used by CloudActions to handle incoming requests and the REST/SOAP protocol which enables the communication of the CloudMessages with the service providers.

Platform Service Registry. The Platform Service Registry, as the name implies, keeps track of the services that the cloud application consumes.

B. Platform Service API Description

The second part in the process of adding a platform service and providers to the framework constitutes the description of the web API. In order to enable the uniform description of the platforms services' API, ontologies are exploited. Specifically, a three level ontological structure, resembling the three lower levels of the Meta-Object-Facility (MOF) standard [9] is defined [10].

The level 2 Ontology (O2) includes the concepts required to describe a web API, such as the operation, the parameters and the endpoint. The level 1 Ontologies (O1), also referred to as Template Ontologies, include the concrete description of each of the platform basic services supported by the framework. In the case of the cloud payment service, information related to charging or refunding a card is captured. The level 0 Ontologies (O0), also known as Instance Ontologies, include the description of the specific platform service providers. A dedicated ontology corresponds to each of the providers and describes the native web API.

1) Platform Service Modelling Phase: During this phase, the platform service reference API, as shown in Fig. 1, is defined. It is exposed to the application developers and describes the operations offered by the particular service. The reference API is captured in the Template Ontology.

Fig. 6 shows a snapshot of the Template ontology for the payment service which describes the operation for charging a card. The name of the operation is "ChargeCard". It is a subclass of the class "Operation". "Operation" is defined in the Abstract platform service ontology (O2 level) and includes all the operations offered by the service. Fig. 7 also includes the following three elements: "CardIdentifier",which denotes the card to be charged, "ChargedAmount", which refers to the amount of money to be charged during the specific transaction and "CurrencyCode" which refers to the currency to be used for the specific transaction. All three elements are subclasses of the class "Attribute". The class "Attribute" is defined in the Abstract platform service ontology and includes all the attributes which are used for the execution of the operations. An attribute is linked to a specific operation with a property. Specifically, the three afore mentioned attributes are linked to the "ChargeCard" operation with the following properties respectively: "hasCardIdentifier", "hasChargedAmount", "hasCurrencyCode".

2) Vendor Implementation Phase: In this phase the provider specific web API is described and mapped to the reference API. The Service API description editor is used to perform the mapping. The outcome is an Instance ontology (O0 level) for each concrete provider.

Particularly, Fig. 7 shows the description of the "charge" operation as defined in the API of the Spreedly service offered via Heroku platform. Individuals are created to express each of the specific elements of the provider`s API. Specifically, the "purchase" Individual denotes the operation name which is equivalent to the "ChargeCard" operation of the Template Ontology. Likewise, the Individual "amount" is of type "ChargedAmount", the "currency_code" is of type "currency" and the "payment_method_token" is of type "CardIdentifier".

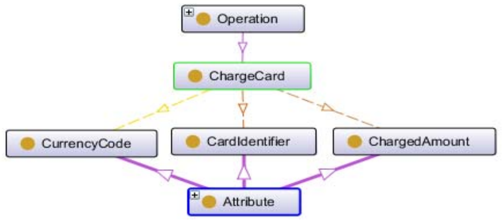


Figure 6. Example of Template ontology for the cloud payment service

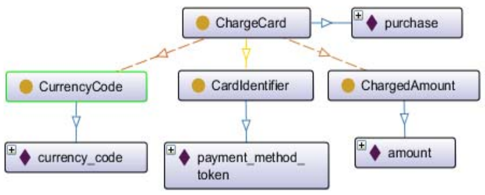


Figure 7. Example of Instance ontology for the Spreedly payment service

In the same way the rest of the functionality of a platform service can be described. At the same time, the differences in the APIs between the various providers are captured. The payment service has been used as an example. The proposed structure of the three levels of ontologies can be used to describe the web API of additional platform services such as authentication and message queue service.

3) Execution Phase: During the Execution Phase, the Platform Service Reference and the provider specific API descriptions, which correspond to the Template and the Instance Ontologies respectively, are fed to the API Client Generator (Fig. 1) [10]. This component parses the Ontologies and generates: (i) A set of interfaces which correspond to the reference API and provide the application developer with access to the functionally of the service. (ii) The client code for the web API invocation of each of the concrete providers which implement the platform service. Therefore the application developers can seamlessly deploy the platform service providers without being required to adhere to the specific web APIs or manually implement the client for each individual API.

IV. CONCLUSIONS

The development framework described in this paper facilitates the integration of platform basic services in a consistent way and the seamless deployment of the concrete providers implementing those services. It achieves this by alleviating the variability across the platform services, namely: (i) the differences in the workflow when executing an operation, (ii) the heterogeneous web API of the providers and (iii) the various configuration settings that each provider requires. The main components of the framework are: (i) the reference meta-model, which enables the modelling of the abstract functionality of the platform basic service and (ii) an ontology-based architecture which alleviates the differences between the Providers' web APIs and automatically generates the client adapters for the API invocation. Finally, the process of adding a platform service provider in the framework was described in three steps: (i) The Platform Service Modelling phase, where the abstract functionality is captured, (ii) the Vendor Implementation phase, where the specific provider functionality is infused and the (iii) Execution phase where the framework handles the operation execution.

V. ACKNOWLEDGMENTS

The research is funded by the EU 7th Framework Programme (FP7/2007-2013) under grant agreement nº264840, the RELATE project (http://www.relate-itn.eu).

VI. REFERENCES

[1] D. Kourtesis, K. Bratanis, D. Bibikas, and I. Paraskakis, "Software Co-development in the Era of Cloud Application Platforms and Ecosystems: The Case of CAST," in *Collaborative Networks in the Internet of Services*, Bournemouth, 2012, pp. 196–204.

[2] F. Gonidis, I. Paraskakis, and A. J. H Simons, "A Development Framework Enabling the Design of Service-Based Cloud Applications," in *2nd International Workshop on Cloud Service Brokerage,* Manchester, 2014, in press.

[3] jclouds. (2014). [Online]. Available: http://www.jclouds.org.

[4] D. Petcu, "Consuming Resources and Services from Multiple Clouds," Journal of Grid Computing, vol. 12, nr. 2, pp. 1-25, Jan 2014.

[5] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann and A. Nowak, "OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications," in *11th International Conference on Service Oriented Computing,* Berlin, 2013, pp. 692-695.

[6] D. Ardagna, E. Di Nitto, G. Casale, D. Petcu, P. Mohagheghi and S. Mosser, "MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds," in *Workshop on Modeling in Software Engineering*, Zurich, 2012, pp. 50-56.

[7] K. Jeffery, G. Horn and L. Schubert, "A vision for better cloud applications," in I*nternational Workshop on Multi-cloud applications and federated clouds*, Prague, 2013, pp.7-12.

[8] R. Fielding, "Architectural styles and the design of network-based software architectures," PhD thesis, University of California, Irvine, 2000.

[9] T. Gardner, C. Griffin, J. Koehler and Rainer Hauser, "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard," in *Workshop on Metamodeling for MDA*, York, 2003, pp. 179–197.

[10] F. Gonidis, I. Paraskakis and A. J. Simons, "On the role of ontologies in the design of service based cloud applications," in *Second Workshop on Dependability and Interoperability in Heterogeneous Clouds,* Porto, 2014, in press.