A Development Framework Enabling the Design of Service-Based Cloud Applications

Fotis Gonidis[1], Iraklis Paraskakis[1], Anthony J. H. Simons [2]

[1] South-East European Research Centre (SEERC),
International Faculty of the University of Sheffield, City College
24 Proxenou Koromila Street, 54622 Thessaloniki, Greece
{fgonidis,iparaskakis}@seerc.org

[2] Department of Computer Science, The University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield S1 4DP, United Kingdom
{A.Simons}@dcs.shef.ac.uk

Abstract. Cloud application platforms gain popularity and have the potential to change the way applications are developed, involving composition of platform basic services. In order to enhance the developer's experience and reduce the barriers in the software development, a new paradigm of cloud application creation should be adopted. According to that developers are enabled to design their applications, leveraging multiple platform basic services, independently from the target application platforms. To this end, this paper proposes a development framework for the design of service-based cloud applications comprising two main components: the meta-model and the Platform Service Manager. The meta-model describes the building blocks which enable the construction of Platform Service Connectors in a uniform way while the Platform Service Manager coordinates the interaction of the application with the concrete service providers and further facilitates the administration of the deployed platform basic services.

Keywords: platform basic services, abstract service models, multi-cloud

1. Introduction

The emergence of the cloud application platforms has been accompanied by a growing number of platform basic services being provisioned via them. In addition to the traditional platform resources such as programming environment and data stores [1], a cloud application platform provisions a range of platform basic services that developers can leverage to accelerate the software development process [2]. A platform basic service, in the Platform as a Service level [1], can be considered as a piece of software which offers certain functionality and can be reused by multiple users. It is typically provisioned via a web Application Programming Interface (API) either REST [3] or SOAP [4]. Examples of such services are the message queue, the e-mail, the authentication and the payment service.

The rise of the platform basic services has the potential to lead to a paradigm of software development where the services act as the building blocks for the creation of service-based cloud applications. Applications do not need to be developed from ground-up but can rather be synthesised from various platform basic services increasing rapidly this way the productivity. This paradigm of software development can be considered as an evolution of the Service Oriented Architecture (SOA) [5] approach, where the applications are composed of various web services. In that case, established frameworks, such as the Business Process Execution Language (BPEL) [6] and the Web Service Resource Framework (WSRF) [7] assist developers during the integration process of the web services. However, the advent of the cloud application platforms and the platform basic services has resulted in multiple software vendors offering the same type of service such as authentication service, mailing service and payment service. Therefore, developers should not only be enabled to effortlessly integrate the platform basic services but also to choose seamlessly the concrete service providers, overcoming the heterogeneity among them.

Towards this direction, a new approach for the design of service-based cloud applications must be adopted. The key concept is for users not to develop applications directly against proprietary cloud provider’s environment. Rather, they should use either standard and widely adopted technologies or abstraction layers which decouple application development from specific target technologies and Application Programming Interfaces (APIs).

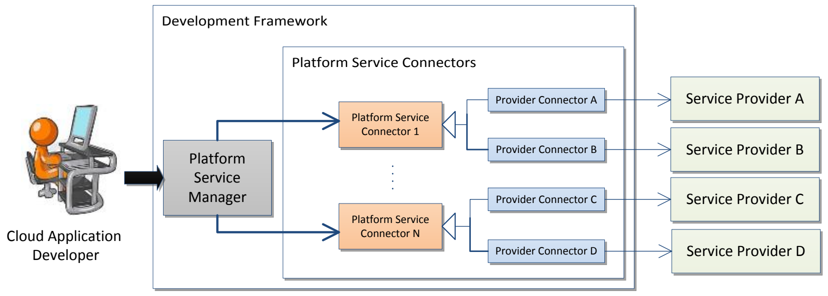


Fig. 1. Cloud Application Development Framework

To this end the paper proposes a development framework which promotes uniform access to platform basic services via the use of abstract Platform Service Connectors (Figure 1). It is composed of three main parts: (i) the Platform Service Manager (PSM), which handles the execution of the services, (ii) the Platform Service Connectors (PSC), which contain an abstract description of the functionality of the services and (iii) the Provider Connectors (PC), which include the detailed implementation required by each provider.

The key objective of the proposed solution is two-fold. First, it introduces a reference meta-model which enables the integration of platform basic services in a consistent way through the construction of the PSCs. Second, it decouples application development from vendor specific implementations by encapsulating the latter in the PCs components. In addition to the reference meta-model, the proposed framework automates the workflow execution of the platform service operations.

The remainder of the paper is structured as follows. The next Section reviews established work in the field. Section 3 describes the way platform basic services may be consumed and motivates the need for a meta-model for constructing the PSCs in a uniform manner. Subsequently, Section 4 states the high-level components of the meta-model and the framework which manages the execution of the PSC. In order to illustrate how the proposed solution can be utilised to enable uniform access to platform basic services, Section 5 illustrates the case of the cloud payment service.

2. Related Work

The constant increase in the offering of platform basic services has resulted in a growing interest in the field of cross platform development and deployment of service-based cloud applications. Significant work has been carried out in the field which can be grouped into three high-level categories: library based solutions [8, 9], middleware platforms [10] and Model-Driven Engineering (MDE) [11] based initiatives [12-15]. Representative work on each of the three categories is presented.

Library-based solutions such as jclouds [8] and LibCloud [9] provide an abstraction layer for accessing specific cloud resources such as compute, storage and message queue. While, library-based approaches efficiently abstracting those resources, they have a limited application scope which makes it difficult to reuse them for accommodating additional services.

Middleware platforms constitute middle layers which decouple application development from directly being developed against specific platform technologies and deployed on specific platforms. Rather, cloud applications are deployed and managed by the middleware platform which has the capacity to exploit multiple cloud platform environments. mOSAIC [10] is such a PaaS solution whose main target is to facilitate the design and execution of scalable component-based applications. The main application building block in the mOSAIC platform is the cloudlet. A platform container manages the cloudlets and has the ability to spawn or destroy instances with respect to the load. Additionally mOSAIC offers an open source API in order to enable the applications to use common cloud resources offered by the target environment such as virtual machines, key/value stores and message queues. mOSAIC adopts a particular programming style based on the cloudlets which impose that applications abide by this style. Thus, although the mOSAIC platform is able to exploit multiple cloud environments, the applications which leverage mOSAIC's benefits, are tightly connected with the specific technology. Furthermore, middleware solutions often are complex environments which may impose an unnecessary overhead, should the applications not exploit all of their features.

Initiatives that utilise MDE techniques present meta-models which can be used for the creation of cloud platform independent applications. The notion in this case is that cloud applications are designed in a platform independent manner and specific technologies are only infused in the models at the last stage of the development. MODAClouds [12] and PaaSage [13] are both FP7 initiatives aiming at cross-deployment of cloud applications. Additionally, they offer monitoring and quality assurance capabilities. They are based on CloudML [16], a modelling language which provides the building blocks for creating applications deployable in multiple IaaS and PaaS environments. Hamdaqa et al. [14] have proposed a reference model for developing applications which make use of the elasticity capability of the cloud infrastructure. Cloud applications are composed of CloudTasks which provide compute, storage, communication and management capabilities. MULTICLAPP [15] is a framework employing MDE techniques during the software development process. Cloud artefacts are the main components that the application consists of. A transformation mechanism is used to generate the platform specific project structure and map the cloud artefacts onto the target platform. Additional adapters are generated each time to map the application's API to the respective platform's resources.

The solutions listed in this Section focus mainly on eliminating the technical restrictions that each platform imposes, enabling this way cross-deployment of cloud applications. Additionally, they offer monitoring and quality assurance capabilities as well as the creation of elastic applications. On the contrary, the vision of the authors is to facilitate the use of platform basic services and concrete providers from the various cloud application platforms in a seamless and transparent manner. To this end, rather than focusing on the obstacles imposed during the deployment of cloud applications we focus on the commonalities and differences exposed by the various platform service providers during the consumption of those by the cloud applications. The proposed solution may be positioned in the intersection of the work presented in this Section. A reference meta-model is introduced to enable the consistent modelling and integration of the various platform basic services such as the authentication, payment, e-mail service. Additionally, a middleware framework handles the execution of the workflow and accommodates the abstraction of the various concrete providers so that application developers are not bound to specific vendor implementations.

3. The Need for a Platform Service Meta-Model

Before describing the proposed framework and the meta-model for constructing the Platform Service Connectors (PSCs), we motivate the need for such a solution. We do so by examining various implementations of platform service clients. Preliminary work of the authors on several platform service providers [17] offered by Heroku [18], Google App engine [19], AWS marketplace [20] have shown that platform services may be distinguished into two categories: stateless and stateful. [21]

Stateless services offer operations which are completed in one step. This means that the user of the service initiates a request and the latter responds with the result of the operation. The requests are performed using the web API exposed by the service providers and usually are in the form of a REST or SOAP call. Examples of such services include the message queue and the e-mail services. For example, in case that the user wants to send an e-mail using an e-mail service provider, he merely needs to submit a web request with a minimum set of required fields: recipient, sender, subject and body. Upon the successful post of the e-mail, the provider responds with a confirmation message.

On the other hand, stateful services require two or more steps in order to complete an operation. Therefore, contrary to the first category, a coordination mechanism is required to handle the operation flow. Additionally, the process involves incoming requests originated either by the client of the application or the service provider and which needs to be handled by the application.

Such an example is the payment service that enables developers to accept payments through their application. In this case the client initiates the purchase flow by sending a request to the application via the user interface. The latter receives the request and subsequently notifies the payment provider about the purchase operation. The provider responds to the application with information regarding the purchase transaction. Afterwards the client fills in the payment card details and transmits the data to the payment provider. Once the validation of the card is completed the provider responds to the application with the result of the payment transaction.

In this process two types of requests are implied. The first one includes the requests performed by the application towards the payment providers and which are executed using the web API offered by the providers. They are the similar to those described in the stateless services. The second type involves incoming requests submitted to the cloud application either by the client or the payment provider and which need to be received and handled by the cloud application.

In addition to the variety of the requests described above, platform basic services in both categories share some common characteristics. Certain configuration settings and credentials are required when a cloud application interacts with a platform service. For example in the case of the payment service, among others, a “redirect URL” needs to be specified to inform the service provider how to perform a request to the application. Regarding the requests performed using the web API of the service provider, authorization information and knowledge of the endpoints are required to execute the web call.

As it became clear a cloud application may interact with several platform basic services in various ways. If we count in the large number of services that an application may be composed of, one can realize that the integration and management of the services may become a time consuming and strenuous process. In order to enable the consistent modelling and integration of services as well as the decoupling from vendor specific implementations, a reference meta-model is required.

The meta-model should be platform and service independent so that it facilitates the design and implementation of a wide range of PSCs. Towards this direction the abstract description of the platform basic service functionality is modelled. Then, the technical details and the specific implementation of each service providers are infused in a transparent to the cloud application manner. Additionally, the Platform Service Manager (Figure 1) keeps track of the platform basic services consumed by the application and coordinates the interaction between the application and the services.

4. The Development Framework

In this Section the high-level components of the development framework are described (Figure 1). This can be further decomposed into (i) the meta-model used to create the Platform Service Connectors (PSCs) and (ii) the Platform Service Manager (PSM) which handles the interaction between the cloud application and the platform service (Figure 2).

4.1 Meta-Model Components

This Section states the components of the meta-model. In essence the meta-model describes the building blocks of which a PSC is composed. As depicted in the lower component of the Figure 2 there are 5 main concepts:

1. **CloudAction.** Cloud Actions are used to model stateful platform basic services as described in Section 3, which define more than one step in order to complete an operation. The whole process required to complete the operation can be modelled as a state machine. Each step can be modelled as a concrete state that the platform service can exist in. When the appropriate event arrives an action is triggered to handle the event and subsequently causes the transition to the next state. The events in this case are the incoming requests arriving either by the application user or the service provider. A separate Cloud Action is defined to handle each incoming request and subsequently signals the transition to the next state.
2. **CloudMessage.** CloudMessages can be used to model requests performed by the cloud application towards the service provider. In this case the web API exposed by the provider is used, usually implemented with the REST or the SOAP protocol. CloudMessages can be used in platform services where the operation can be completed in one step, namely one REST/SOAP request to the service provider. Example of such a request, as mentioned in the previous section, is the e-mail service. A CloudMessage can be defined to send the web request along with the required fields: recipient, sender, title and body. In addition, CloudMessages can be used within Cloud Actions when the latter are required to submit a request to the service provider.
3. **PlatformServiceStates.** The PlatformServiceStates description file holds information about the states involved in an operation and the corresponding Cloud Actions which are initialised to execute the behavior required in each state. A part of a state description file describing the states involved in the payment transaction of a particular service provider is shown here:

```
<StateMachine>
  <State name="PaymentForm"
    action="org.paymentservice.FillOutFormCloudAction"
    nextState="SendTransaction" />
  <State name="SendTransaction"
    action="org.paymentservice.SendTransactionCloudAction"
    nextState="Finish" />
</StateMachine>
```

Two states are described here. For each state the following information is provided: a) The name of the state, b) The CloudAction which needs to be initialised in order to handle the incoming requests and c) the next state which follows when the action finishes the execution. The state named "Finish" signals the completion of the operation.

4. **ConfigurationData.** Certain configuration settings are required by each platform service provider. That information is captured in the ConfigurationData. Example of settings which needs to be defined are the clients' credentials required to perform web requests and the redirect URL parameter which is often requested by the service provider in order to perform requests to the cloud application.
5. **API Service Description File.** The API service description file describes the functionality offered by the service provider via the web interface. The concrete operations, parameters and endpoints are stated in the file. It is consumed by the framework in order generate the client adapter which is used by the CloudMessages to communicate with the service provider.

The concepts listed in this Section enable the modelling of the PSCs and contribute to the first objective of the proposed solution which is to facilitate the integration of platform basic services in a consistent way. Additionally, the consistent modelling of the PSC enables the automation of the workflow execution of the platform service operations.

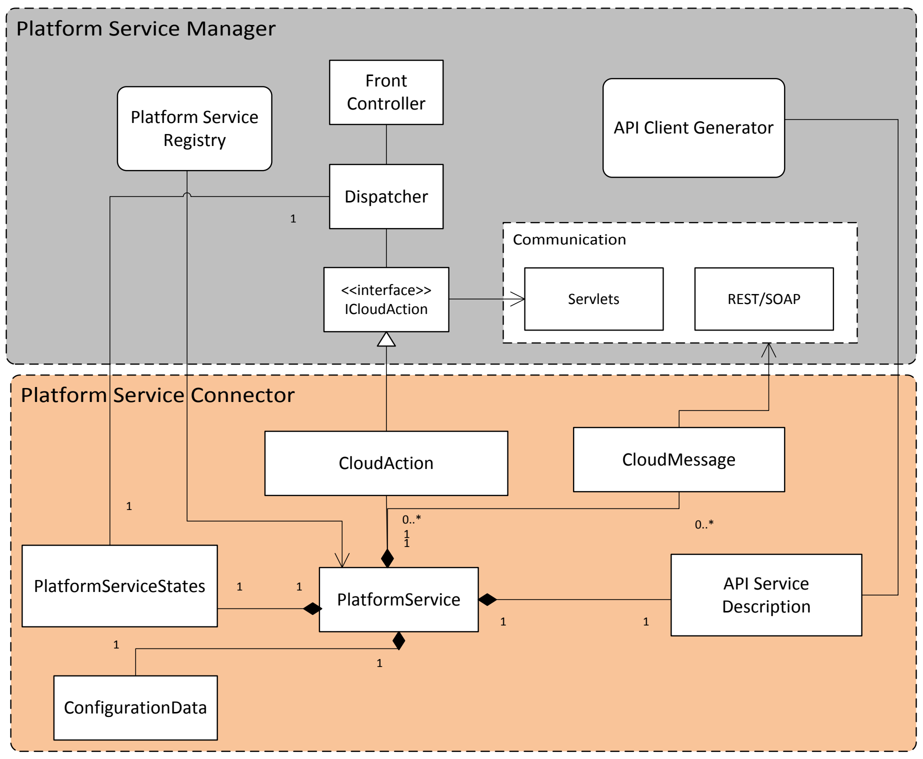


Fig. 2. High level overview of the development framework

4.2 Framework Components

In this Section the high level components comprising the PSM, handling the PSCs, are described. As seen in the upper part of the Figure 2, it essentially consists of the following components:

1) **Front Controller.** The Front Controller [22] serves as the entry point to the framework. It receives the incoming requests by the application user and the service provider.
2) **Dispatcher.** The dispatcher [23] follows the well-known request-dispatcher design pattern. It is responsible for receiving the incoming requests from the Front Controller and forwarding them to the appropriate handler, through the ICloudAction which is explained below. As mentioned in 3.1, the requests are handled by the CloudActions. Therefore the dispatcher forwards the request to the proper CloudAction. In order to do so, he gains access to the platform service states description file and based on the current state it triggers the corresponding action.
3) **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.
4) **Communication patterns.** Two types of communication pattern are supported by the framework: The first one is the Servlets and particularly the Http Servlet Request and Response objects [23] which are used by the CloudActions in order to handle incoming requests and respond back to the caller. The second type of communication is via the use of the REST/SOAP protocol which enable the CloudMessages to perform external requests to the service providers.
5) **Cloud Service Registry.** The Cloud Service Registry, as the name implies, keeps track of the services that the cloud application consumes.
6) **API Client Generator.** Based on the API Service Description file, the API client generator maps the provider’s specific API to the abstract one defined in the PSC. In case the provider offers additional functionality, the respective client is updated. The updated client is used by the CloudMessages to communicate with the service provider.

The components of the framework listed in this Section facilitate the workflow execution of the platform service operations and further automate the generation of the Web API clients required to interact with the platform services. Along with the PSCs, they contribute to the second objective of the proposed solution which is to decouple the cloud application from directly interacting with the vendor specific implementations and thus enabling developers to choose seamlessly the concrete service providers.

5. The Case of the Cloud Payment Service

In order to illustrate how the meta-model and the Platform Service Manager (PSM) can be utilised to facilitate the consumption of platform basic services by the applications, the case of the cloud payment service is presented. The payment service enables a website or an application to accept online payments via electronic cards such as credit or debit cards. The added value that such a service offers is that it relieves the developers from handling electronic payments and keeping track of the transactions. The payment provider undertakes the task to verify the payment and subsequently informs the application about the outcome of the transaction. The payment service has been chosen because of its inherent relative complexity compared to other services such as e-mail or message queue service. The complexity lies in the fact that the purchase transaction requires more than one state to be completed and there is a significant heterogeneity among the available payment providers with respect to the involved states.

In order to enable the cloud application developer to choose seamlessly the optimal payment provider, the various provider implementations need to be modelled and added to the framework so that the latter can handle the flow of the operations. This way the application developers are relieved from implementing explicitly the interactions with each payment provider.

The process can be divided into three steps:

1) Modelling of the states of the cloud payment service. Several payment service providers need to be studied in order to extract a common state chart capturing the operation flow.
2) Based on the state chart constructed in the previous step, a model is created utilising the meta-model described in Section 4.
3) Capturing of the provider specific data and mapping on the abstract model built in step 2.

5.1 State Modelling of the Platform Service

The first step towards modelling the states of the cloud payment service is to explore the concrete payment providers and extrapolate the common states in which they may co-exist. For that reason 9 major payment service providers have been studied [24-32], provisioned either via a major cloud platform such as Google App Engine and Amazon AWS or via platform service marketplaces such as Heroku add-ons and Engineyard add-ons. These providers can be grouped into three main categories. An exhaustive listing of the characteristics of each payment provider is out of the scope of this paper. Rather, we focus on demonstrating how concrete providers can be mapped on the abstract model. Therefore, in this paper we present the case of one category, the “transparent redirect” and use as the concrete payment provider, the Spreedly [30], a payment provider offered via Heroku platform.

Transparent redirect is a technique deployed by certain payment providers in which, during a purchase transaction, the client’s card details are redirected to the provider who consequently notifies the cloud application about the outcome of the transaction.

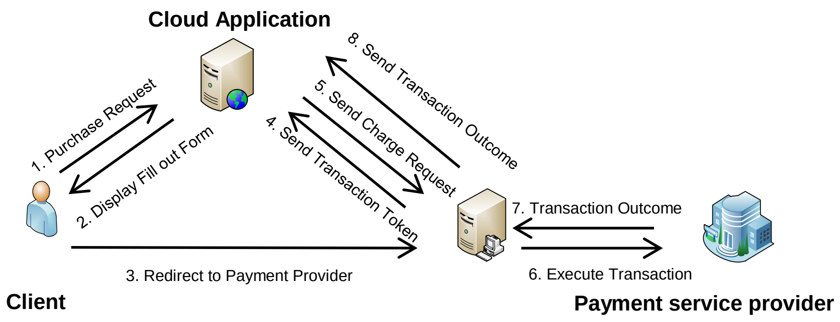


Fig. 3. Cloud Payment Service

Figure 3 describes the steps involved in completing a payment transaction, while Figure 4 shows the state chart of the cloud application throughout the transaction. Two states are observed. While the cloud application remains in the first state, it waits for a payment request. Once the client requests a new payment, the cloud application should display the fill out form where the user enters the payment details.

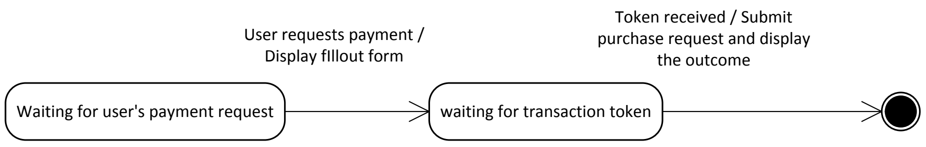

User requests payment /
Display fIllout form
Token received / Submit
purchase request and display
the outcome
Waiting for user's payment request
waiting for transaction token

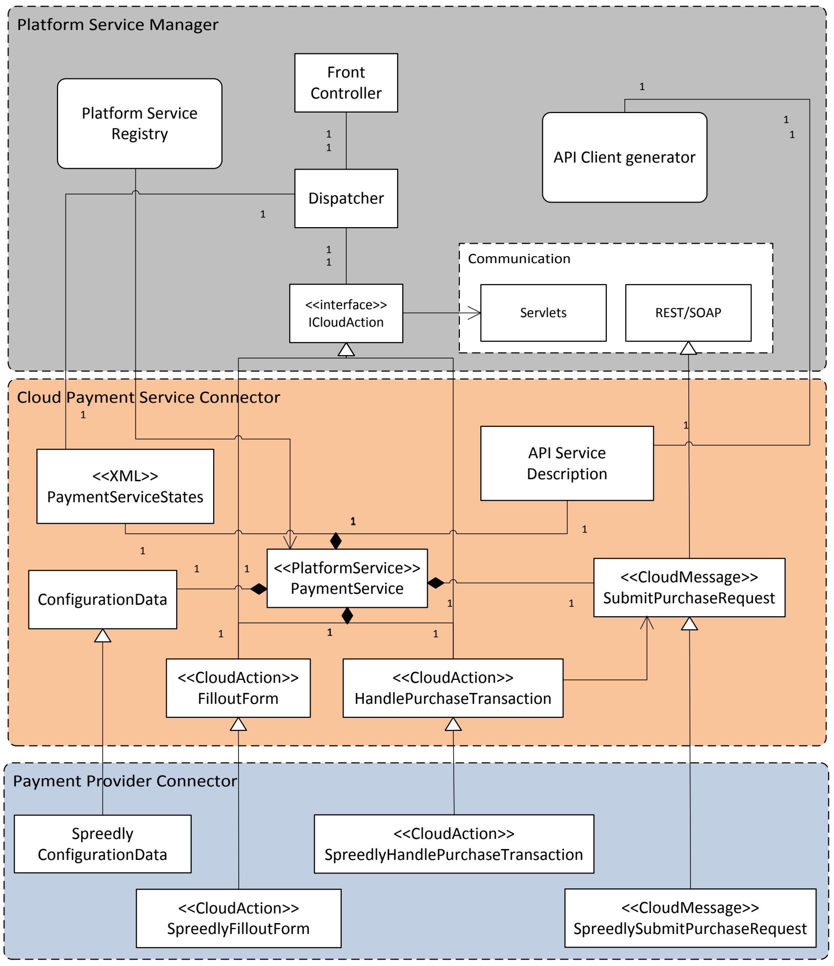
Fig. 4. State chart of the cloud payment service

Subsequently, the cloud application moves to the next state where it waits for the transaction token issued by the payment provider. The transaction token uniquely identifies the current transaction and can be used by the cloud application to complete the purchase. Once the user submits the form, she is redirected to the payment provider who validates the card details. Then a request to the cloud application is submitted including the transaction token. Once the token is received the application submits a request to the provider with the specific amount to be charged. The provider completes the transaction and responds with the outcome. Depending on the outcome, the cloud application displays a success or failure page to the client.

5.2 Mapping of the State Model on the Meta-Model

Based on the state chart mentioned in the previous Section a provider independent model is constructed using as building blocks the meta-model described in Section 4. The model is constructed as follows:

1) For each state where the application waits for an external request, a CloudAction is defined to handle the request.
2) For each request initiated by the cloud application targeting the service provider a CloudMessage is defined.



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

Fig. 5. Cloud Payment Service Model

As seen in Figure 5, the following blocks are defined:

a. FilloutForm. The FilloutForm receives the request for a new purchase transaction and responds to the client with the fill out form in order for the latter to enter the card details. The communication is realised using the servlet technology.

b. HandlePurchaseTransaction. The HandlePurchaseTransaction receives the request from the service provider containing the transaction token. Then, a request is submitted to the provider including the transaction token and the amount to be charged. The provider replies with the outcome of the purchase and subsequently the action responds to the client with a success or fail message accordingly.

c. SubmitPurchaseRequest. The SubmitPurchaseRequest is a CloudMessage used internally by the HandlePurchaseTransaction action. Its purpose is to model the request to the service provider, using the exposed web API, to complete the purchase transaction. It receives the provider’s respond stating the outcome and forwards it to the action.

d. ConfigurationData. The ConfigurationData contains the service settings required to complete the purchase operation. Particularly, the following piece of information is listed: the “redirectUrl”, the username and the password.

e. PaymentSerivceStates. In the PaymentServiceStates file the states and the corresponding actions involved in the transaction are defined. The file is used by the framework to guide the execution of the actions.

At this point the Platform Service Connector 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.

5.3 Mapping the Provider Specific Implementation on the Abstract Model

After having defined the generic model for the payment service, the concrete implementation and settings for the providers needs to be infused. For each CloudAction and CloudMessage defined in the model in Figure 5, the respective provider specific blocks should also be defined, namely the: SpreedlyFilloutForm, SpreedlyHandlePurchaseTransaction and the SpreedlySubmitPurchaseRequest. In addition, the ConifgurationData file and the API service description needs to be updated accordingly to match the specific provider. The final step is to declare the concrete actions to be triggered in the Payment Service States file.

Should the provider’s implementation accurately matches the model, the provider specific Actions and Messages can reuse the functionality of the generic model. In case the provider’s implementation diverts from the generic model the model’s functionality can be overridden.

The process described in this Section constitutes a method towards enabling the platform basic services to be modelled in a consistent manner. Subsequently, the proposed management framework handles the interaction between the cloud application and the specific platform service providers. The framework is continuously enriched with additional service Providers Connectors. In case certain providers cannot be accommodated by the existing PSC models, additional custom CloudActions and CloudMessages can be defined.

6. Conclusions

This paper proposed a development framework and a meta-model for designing service-based cloud applications. Platform basic services are becoming increasingly popular and have the potential to act as building blocks for the development of applications. As a result, developers should be enabled to integrate platform basic services in a consistent way and choose seamlessly the concrete service providers.

Towards this direction, the meta-model presented in Section 4 expedites the modelling of abstract Platform Service Connectors. The latter constitutes the intermediate layer between the cloud application and the concrete service Provider Connectors. The main components of the meta-model are the CloudActions and the CloudMessages. The former facilitates the modelling of the incoming requests which needs to be handled by the application, while the latter are used for the requests initiated by the application targeting the service providers. The case of the cloud payment service illustrated how the proposed solution can facilitate the modelling of the platform basic services and accommodate concrete service providers.

In addition, the Platform Service Manager described in this work coordinates the interaction between the application and the service providers. At the same time it paves the way for an integrated solution which enables the application developers efficiently managing the platform basic services they consume. Future work involves refining the components of the framework such as the API Client Generator and the PlatformService Registry and applying the proposed solution to a variety of platform basic services.

Acknowledgment. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement nº264840, the RELATE project (http://www.relate-itn.eu).

References

1. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. National Institute of Standards and Technology, vol. 53, no.6, p.50 (2009)
2. Kourtesis, D., Bratanis, K., Bibikas, D., Paraskakis, I.: Software Co-development in the Era of Cloud Application Platforms and Ecosystems: The Case of CAST. In Collaborative Networks in the Internet of Services, pp. 196–204. Bournemouth, UK, (2012)
3. Fielding, RT., (2000). The REpresentational State Transfer (REST). PhD dissertation, Irvine: Department of Information and Computer Science, University of California. [Online]. Available: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm
4. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., Winer, D., (2000). Simple Object Access Protocol (SOAP) 1.1. [Online]. Available: http://www.w3.org/TR/SOAP/
5. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
6. Andrews, T., Curbera, F., Dholakia, H., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services Version 1.1. Technical report (2003), xml.coverpages.org/BPELv11-20030505-20030331-Diffs.pdf
7. Web Services Resources Framework (WSRF 1.2). Technical Report, OASIS (2006), https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
8. jclouds. (2014). [Online]. Available: http://www.jclouds.org
9. Apache LibCloud. (2014). [Online]. Available: https://libcloud.apache.org/index.html
10. Petcu, D.: Consuming Resources and Services from Multiple Clouds. Journal of Grid Computing, nr. 10723, pp 1-25. Jan (2014)

11. Kent, S.: Model Driven Engineering. In: Third International Conference on Integrated Formal Methods, pp.286-298. Turku, Finland (2002)
12. Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi P., Mosser, S., Matthews, P. , Gericke, A., Ballagny, C., D'Andria, F., Nechifor, C. S., Sheridan, C.: MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. In : Workshop on Modeling in Software Engineering, Zurich, Switzerland (2012)
13. Jeffery, K., Horn, G., Schubert, L.: A vision for better cloud applications. In: Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds, pp. 7-12. Prague, Czech Republic (2013)
14. Hamdaqa, M., Livogiannis, T., Tahvildari, L.: A reference model for developing cloud applications. In: 1st International Conference on Cloud Computing and Services Science, pp. 98-103, Noordwijkerhout, The Netherlands (2011)
15. Guillen, J., Miranda, J., Murillo. J. M., Cana, C.: Developing migratable multicloud applications based on MDE and adaptation techniques. In: the 2nd Nordic Symposium on Cloud Computing & Internet Technologies, pp. 30-37. Oslo, Norway (2013)
16. Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A.: Managing multi-cloud systems with CloudMF. In: the 2nd Nordic Symposium on Cloud Computing & Internet Technologies, pp. 38-45. Oslo, Norway (2013)
17. Gonidis, F.: Experimentation and Categorisation of Cloud Application Platform Services. SEERC Technical Report. Thessaloniki, Greece: South East European Research Centre (SEERC) (2013)
18. Heroku. (2014). [Online]. Available: http://heroku.com
19. Google App Engine. (2014). [Online]. Available: https://developers.google.com/appengine
20. AWS Marketplace. (2014). [Online]. Available: https://aws.amazon.com/marketplace
21. Pautasso, S., Zimmermann O., Leymann F.: Restful web services vs. "big"' web services: making the right architectural decision. In: 17th International Conference on World Wide Web, pp. 805-814. ACM, NewYork (2008)
22. Alur, D., Crupi J., Malks D.: Core J2EE Patterns. Sun Microsystems Press. (2001)
23. Hunter, J., Crawford, W.: Java Servlet Programming, O'Reilly & Associates, Inc., Sebastopol, CA (2001)
24. Amazon Flexible Payments. (2014). [Online]. Available : https://payments.amazon.com/developer
25. AuthorizeNET. (2014). [Online]. Available: http://developer.authorize.net/api/sim/
26. Braintree. (2014). [Online]. Available: http://chargify.com/
27. Chargify. (2014). [Online]. Available: https://www.braintreepayments.com/
28. Google Wallet For Digital Goods. (2014). [Online]. Available: https://developers.google.com/wallet/digital/
29. Paypal Express Checkout. (2014). [Online]. Available: https://www.paypal.com/gr/webapps/mpp/express-checkout
30. Spreedly. (2014). [Online]. Available: https://spreedly.com/
31. Stripe. (2014). [Online]. Available: https://stripe.com/
32. Viva Payment Services. (2014). [Online]. Available: https://www.vivapayments.com/en/