Open Architecture for Developing Multitenant Software-as-a-Service Applications

Javier Espadas, David Concha

Tecnológico de Monterrey, Campus Monterrey
Monterrey, México
{mijail.espadas,david.concha}@itesm.mx

David Romero, Arturo Molina

Tecnológico de Monterrey, Campus Ciudad de México
City, México
david.romero.diaz@gmail.com, armolina@itesm.mx

Abstract. **As cloud computing infrastructures are growing, in terms of usage, its requirements about software design, management and deployment are increasing as well. Software-as-a-Service (SaaS) platforms play a key role within this cloud environment. SaaS, as a part of the cloud offer, allows to the software providers to deploy and manage their own applications in the clouds in a subscription basis. The problem with the current SaaS offers is the lack of openness of in their platforms and the need for learning a whole new paradigm when trying to initiate in the SaaS market. Big players, such as: Amazon, Google or Microsoft, offer their proprietary SaaS solutions. Another consideration is the amount of current Web applications that need to be re-engineered into this cloud paradigm. This research work aims to reduce the effort required to enter into the SaaS market by presenting an architecture based on open source components for developing, deploying and managing SaaS applications.**

Keywords - cloud computing; software-as-a-service; software architecture; open source.

I. INTRODUCTION

Software-as-a-Service (SaaS) has become the new buzz-word around software industry. From a successful business such as Salesforce.com towards new SaaS software architectures with legacy solutions [3], SaaS solutions have been converted into state-of-the-art technology. In spite of the growth of this industry, there is a lack of established software architectures that enable the delivery of business applications as services. Big players (e.g., Microsoft, Google, Amazon) have developed their own SaaS infrastructure in order to deliver their next-generation software applications. As the number and scale of cloud-computing systems continues to grow, significant research is required to determine the strategy towards the goal for making future cloud computing platforms successful. Currently, most cloud-computing offerings are either proprietary or depend on software that is not amenable to experimentation or instrumentation [1][3][4].

New Internet-enabled platforms have appeared, thus enabling open collaboration and creation. These platforms represent a new way of delivering software applications [2][3]. While the practice of outsourcing business functions such as payroll has been around for decades, its realization as an online software service has until recently became popular. In the online service model, the provider develops an application and also operates the servers that host it. Customers access the application over the Internet using industry-standard browsers or Web Services clients [4][6]. Online software delivery is now conceived and defined as Software-as-Service (SaaS). SaaS has become a well established phenomenon in some areas of enterprise IT. It is growing into a mainstream option for software-based solutions and this will impact most of the enterprise IT departments over the next three years [9]. Chou [5] declares that SaaS is the next step in the software industry, not because it is a "cool idea", but because it fundamentally alters the economics of software.

A wide range of online applications, including e-mail, human resources, business analytics, customer relationship and enterprise planning, are available [6]. According to Gartner [8], the SaaS market will be growing in the next years, by 2009 100% of tier 1 consulting firms will have a SaaS practice and by 2011, 25% of new business software will be delivered as SaaS. Also, IDC estimates customers spending on SaaS solutions will increase to $14.8 billion by 2011 [11]; two out of three businesses are either buying or considering buying software via the subscription model [10] and McKinsey reports that the proportion of CIOs considering adoption of SaaS applications in the coming year has grown from 38% a year to 61% [7]. With previous business facts, is possible to realize the importance and quantity of software that will be delivered throughout SaaS environments.

Unfortunately, several SaaS providers offer their own architecture and their own implementation requirements. Salesforce.com, for example, provides the Force.com development platform and it uses a proprietary development model (custom classes and user interfaces) for building SaaS applications. Furthermore, transition from current Web applications or Application Service Providers (ASPs) solutions to this development model is not a trivial task. Concha, et al. [3] and Espadas, et al. [4] identify a number of steps about transitioning from an ASP solution to a SaaS implementation:

- Current ASPs define a single static revenue models (e.g., embedded & hard-coded within the application implementation). When the dynamic nature of markets asks ASPs to modify their revenue model, ASPs are not able to change it in a cost-effective way, mainly because the revenue model is hard-coded into the application.

 ISBN: 978-1-61208-106-9

- Traditional ASPs provide a portal mechanism for accessing their applications. The current implementation of ASP only supports the notion of one service provider. This is the host platform it-self. The benefits of shifting to a multi-provider approach include an easy integration with associates that complement the platform administrator. Migrating to a multiple provider with multiple e-services also provides the ability to deploy and manage independent sets of applications.
- Presently, ASP services are designed, developed and deployed as Web applications. They are managed by the platform through a Web container and there is no other support for them (such as: billing, monitoring, customization, etc.). In other words, we could see SaaS applications as desktop applications running within an operating system.

This research work addresses these issues, by proposing an open architecture for achieving an implementation capable of deploying applications over the Internet on the service premise. This paper is structured as follows: Section II describes the software architecture and core technologies of the SaaS platform; Section III outlines a set of business services that support SaaS applications; Section IV defines a SaaS application and its components; Section V explains the SaaS core application that is used to access to the platform and Sections VI and VII describe multitenant implementation of applications and subscriptions.

II. SAAS ARCHITECTURE

The architecture bases the communication layer on a Service Oriented Architecture (SOA) that supports techniques for constructing reliable services on cloud computing infrastructures [15].

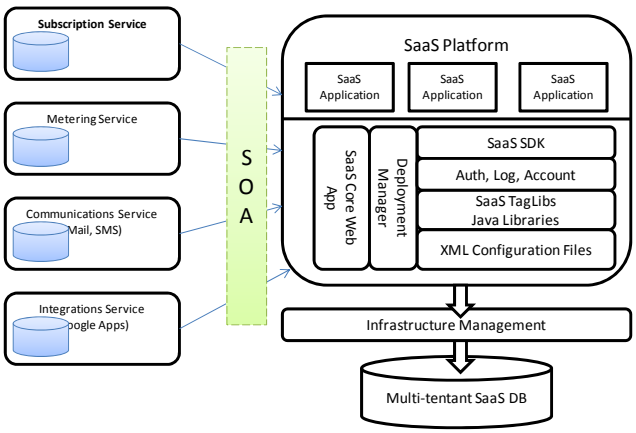

Figure 1. SaaS architecture

The SaaS platform is composed of several components that allow the deployment of applications as services (Fig. 1). Each component is integrated in an Apache Tomcat container as a Web application (.war), a packaged library (.jar) or a business services (Web application + Web Services). A '*service application*' is defined as the application that will be delivered as a service to the customers. Each service application is deployed as a common Web application within the Tomcat container and it manages its own resources, such as data sources, libraries, and views. The main difference with common Web deployments is about how the SaaS components manage and interact with these Web applications. The main interaction point of the service application with the platform is done through a SaaS Application Programming Interface (API). The SaaS API provides the common libraries that are used by the applications to access the basic SaaS services, such as: authentication, account information, public resources and so on. In the view layer, the platform offers components (SaaS Tag Libraries) for an easy integration with the SaaS context (such as: public/private menus, templates, layouts). The Deployment Manager is a listener component that configures each application according to its configuration file (*appService.xml*). Every time a Web application is deployed within the Tomcat container, the Deployment Manager reads the configuration file and analyzes the application code for detect updated or new modules, security roles or deployment changes. The access point to the SaaS platform is the SaaS Core Web Application (SCWA). This component is a Web application that is used to access to all other applications and components. SCWA is in charge of loading common resources and views, such as security context, authenticated user, view filters, etc.

TABLE I. OPEN SAAS PLATFORM TECHNOLOGIES

Requirement	**Technology**
Language Platform	J2EE (Java 1.6)
Web Container	Apache Tomcat 6
Web Framework	Struts 2
Web Services	Apache Axis2
Dependency Injection	Spring 2
Dependency Injection + Web Services integration	WSO2
Multi-tenancy Layer	JoSQL + Java Annotations
Persistence Layer	Hibernate 3 and Java Persistence API (JPA)
Database Management	MySQL 5

As Table I outlines, the core technologies of the SaaS implementation are open source projects. In Figure 1 it is possible to find a set of business services that are consumed through a platform. These business services were designed, developed and deployed by following a Service Oriented Architecture (SOA) design in order to be completely decoupled to the SaaS platform. Each business component exposes a set of Web Services that can be consumed through the platform (or even others platforms) as a client. But this schema can be bidirectional; a business component can be a client of the platform as well. The implementation of these business services will be explained in a further section.

 ISBN: 978-1-61208-106-9

III. BUSINESS SERVICES

A set of support business services is available for service applications. As such, the SaaS applications do not implement code for these mechanisms as they are provided by the platform. The implemented services are:

- *Metering & monitoring.* SaaS platform provides automatic and non-intrusive support for metering applications and tenant-based monitoring.
- *Mailing.* A component for sending/managing electronic mail within applications without complex configuration and programming.
- *Application customization.* The customization component allows the subscriber to customize their own data by adding fields to their business objects (e.g., contact, lead, bill, etc.). By adding custom fields to business object it is possible to generate personalized capture and search forms and to create filters for these custom properties.

Each business component is developed as a Web application, but it exposes a set of Web services through WSO2 framework [16], which integrates web services deployed through Apache Axis2 and dependency injection with Spring 2. Each business component application implements its own Web services and they are referenced in the *applicationContext.xml* Spring file. In this way, any application in the platform can expose its own Web services through simple classes, without having to implement complex mechanism to generate WSDL documents. Same implementations were followed for other business services (e.g., metering, subscriptions, customization). Though, the business services implement other functionalities for their own management and configuration. For example, Mail Service application offers the possibility for configure manage their mail queue and the providers' mail accounts.

IV. SAAS DEVELOPMENT & DEPLOYMENT

A SaaS application is a Web application deployed within the SaaS platform with a particular configuration. A service application is a set of Web components that can be seen as a whole software application. It provides a set of functions separated by modules that can be deployed on demand into a SaaS platform. In a simple way a service application only has:

- *Views.* All the screens and forms that the user can interact with.
- *Business Logic.* Code for actions, business logic and data source accesses.
- *Configuration files.* XML or properties files.
- *Database.* Storage for application data; logically separated for each subscriber.

That is, the SaaS application must not implement code for authentication and authorization, application metering, customization, etc, because they must be provided by the SaaS platform (as described in Section III). In the SaaS platform, the applications' services are developed and deployed as common Web applications but with specific features and configurations that are interpreted by the platform in order to create a SaaS execution environment. The common frameworks and libraries used to develop SaaS applications in the platform are the same as outlined in Table 1. As stated, each application manages its own data sources (e.g., databases, Web services, etc.). The SaaS platform automatically detects configuration such as: business modules, roles, menus, permissions, etc. Once the Web application is deployed and configured, it can be offered as a SaaS solution to multiple customers through a subscription basis. The principal configuration file for deploying a Web application as a SaaS application is the *appService.xml* file. It defines the principal information of a SaaS application.

```
<?xml version="1.0" encoding="UTF-8"?>
<appService>
        <name>Contact Manager</name>
        <label>menu.contactapp</label>
        <version>1.0</version>
        <description>...</description>
        <defaultProvider>TGHEWFS</defaultProvider>
<Role name="manager" description="...">
<Menu>
<MenuItem label="Contacts" path="/contacts/view.action"/>
<MenuItem label="Configure" path="/config/view.action"/>
</Menu>
</Role>
<Role name="user" description="...">
<Menu>
<MenuItem label="Contacts" path="/contacts/view.action"/>
</Menu>
</Role>
</appService>
```

In the last snippet, the XML tag *appService* encloses a set of attributes for the application, such as name, label, description and the default provider which owns the application. The *Role* tag specifies available roles for the application permissions. These roles are updated in the platform database when the platform is initialized in the application server. Within these role tags it is possible to specify application's menus that are presented when the authenticated user has such role. A SaaS application is packaged as a *.war* Java component.

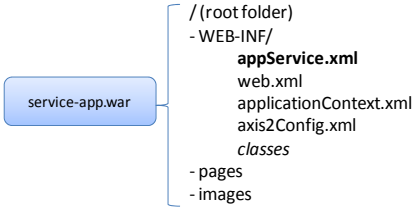

Figure 2. Folder structure of a SaaS application

Figure 2 shows the structure followed by any Web application that is deployed as a SaaS application. This structure shows the location of *appService.xml* file in order to be recognized as a SaaS application by the platform.

 ISBN: 978-1-61208-106-9

The following steps are performed during each SaaS application initialization:

1. A platform component called Deployment Manager reads the *appService.xml* file. This component retrieves information from the service, such as name, version, etc. It inserts or updates the application information in the platform database.
2. Deployment Manager reads the *appService.xml* to create or update the application roles.
3. Deployment Manager inspects the application code for Action classes. This inspection looks up all Java packages that end with '*.actions*' and the classes whose name ends with '*Action*'. For example:
   - *com.myapplication.actions.ContactsAction*
   - *com.myotherapplication.actions.SomeOtherAction*

   These action classes will be inserted or updated in the platform database as modules.
4. Platform inspects each method of a Action class (module). With the use of the *@SaaSFunction* Java annotation it is possible to define functions for each module. This function declaration allows having a method-level granularity about restricted access for each application role.
5. Both modules and functions are synchronized with the platform database.

V. SAAS CORE WEB APPLICATION

The access point for the whole SaaS platform and its deployed applications is known as SaaS Core Web App (SCWA). This component is a Web application with specific characteristics for managing tenant-based authentication, security and control access lists. Each user belongs to one or more subscriber or tenant (these terms will be used indistinctly). Once the user has been authenticated through an email and password, SCWA links the user to its subscriber ID. If the user belongs to two or more subscribers, a selection screen is displayed to select which to work with. After that, SCWA searches for the user within an Access Control List (ACL) to retrieve its permissions for that subscriber and for the SaaS applications that the subscriber has contracted. Then SCWA forms a session cookie with all this information and stores it within the user session. In this way, each user is linked to this tenant-based information and all subsequent requests are identified by this session cookie. However, it is necessary to retrieve this information from several service applications, even when deployed in different machines over a cluster. In order to achieve this, all Web applications should have access to the SCWA and should retrieve the cookies from it. The tenant-based information is stored in the SCWA context session and the rest of the applications can access it through the following mechanism:

```
public static UserVO getAuthenticatedUser() throws
NotAuthenticatedUserException {
        HttpServletRequest request =
ServletActionContext.getRequest();
        String SAASADMIN_SESSIONID =
getCookieValue(request,AuthConstants.SIDEL_SESSION_ID);
        ServletContext contextAuth =
request.getSession().getServletContext();
        UserVO authUser =
getUserFromAdminContext(contextAuth,SAASADMIN_SESSIONID
);
if (authUser==null){throw new NotAuthenticatedUserException();}
        return authUser; }
private static UserVO getUserFromAdminContext(ServletContext
context, String ssosessionid) {
        ServletContext sidelcontext =
context.getContext(SAAS_CORE_APP );
         Hashtable<String, UserVO> shareddata =
(Hashtable<String, UserVO>)sidelcontext.getAttribute(
AuthConstants.SAAS_USERS );
if (shareddata!=null && ssosessionid!=null) {
            // get the right User using the sessionid
            return (UserVO)shareddata.get(ssosessionid);}
          else  return null;
}
```

The static method *getAuthenticatedUser()* can be called from any application and it retrieves the session cookie of the authenticated user from the SCWA context (represented by *SAAS_CORE_APP* variable). *UserVO* is the value object that holds information about the subscriber and the authenticated user.

VI. PERSISTENCE MULTITENANT IMPLEMENTATION

There are different mechanisms for supporting multi-tenancy. The applications services deployed within our SaaS platform implements a *Shared Database - Shared Schemas* mechanism [12][13][14], by separating (logically) the data corresponding to each tenant with a subscriber ID field in the database's tables. This shared schema approach has the lowest hardware and backup costs, because it allows serving the largest number of tenants per database server [14]. As described, each service application implements its own database, separating the multitenant information with a subscriber ID key. An example of a multitenant data model is showed in Figure 3:

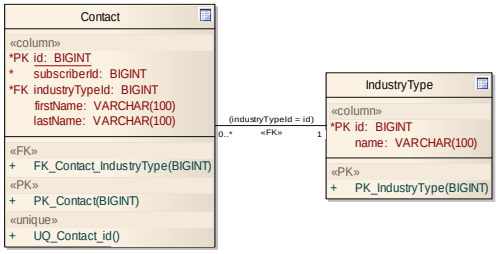

Figure 3. Shared schema for a SaaS application

In Figure 3, It is depicted a shared database scheme for a Contact Manager SaaS application. The core entity is the Contact and each subscriber manages its own set of contacts. As Figure 3 shows, this table has the '*subscriberId*' field; it means that Contact is a tenant-based object. When users log in to the SaaS platform and access to the Contact Manager application, they are allowed to access only to the contacts of their subscribers. As described in the previous section, this subscriber identifier can be retrieved from the SCWA context by any other service application, in this case the Contact Manager. Within the business logic of the application it is possible to retrieve this subscriber ID

 ISBN: 978-1-61208-106-9

and filter the contacts in the persistence layer. The SaaS platform uses an object-oriented mechanism for multi-tenancy which is implemented in the application side and it is called Multi-Tenant Persistence layer. This layer uses JoSQL [17], a LINQ-like [18] technology for perform SQL-like queries over collections and the ability of Struts2 to create Aspect-Oriented interceptors that allows to separate in a logical way the information of each subscriber, supposing the need to retrieve the contacts from a given subscriber. The persistent layer is based on Object Relational Mapping technologies (JPA + Hibernate). We can use a simple object-oriented query to do that:

```
// JPA query in the persistence layer
String sql = "SELECT contact FROM Contact contact"
Query query =  em.createQuery(sql);
return query.getResultList();
```

Simple as is, it is important to notice that there is no filter by subscriber in the query sentence. The persistence layer will return a set of 'Contact' objects. By using the interceptor feature of Struts2 is possible to pre-process these results before they can be accessed by the presentation layer. Within a Struts2 action we can declare an annotated property:

```
@Multitenant(attribute="subscriberId")
List<Contact> contacts = //get contacts from persistence layer
```

The previous code declares that this particular list of objects will be filtered before they are accessible from another component of the application (a Java Server Page view for example). This pre-processing implementation is achieved by setting a Struts2 interceptor in the call stack. This interceptor can access to the invocation action:

```
Object action = invocation.getProxy().getAction();
//getting the subscriber ID from the authentication context
long subscriberId = Auth.getSubscriberId();
for (Field field : clazz.getDeclaredFields() ){
        if (field.isAnnotationPresent(Multitenant.class)){
        Multitenant filter =
(Multitenant)field.getAnnotation(Multitenant.class);
String attribute = filter.getAttribute();
String property = field.getName() ;
Object objList = BeanUtils.getProperty(action, property);    String
className = getClassName( objList );
Query q = new Query ();// create and perform a query over the  list
q.parse("SELECT * FROM "+className+" WHERE "+attribute+"
= "+subscriberId);
QueryResults qr = q.execute (list);
List newList = qr.getResults();
//setting back the filtered list by tenant
BeanUtils.setProperty(action,property,newList);
}}
```

In the example, the 'Contacts' list will be reduced to only the objects which their "*subscriberId*" property matches with the authenticated subscriber. With this mechanism it is possible to create multi-tenant pre-processing behavior within the SaaS applications. In fact, it is feasible to create a transparent support for multi-tenant persistence without affecting the on-premise applications.

VII. MULTITENANT SUBSCRIPTIONS

Basically, the subscription service is a Web application. It uses the same open technologies as the SaaS platform (see Table 1). Its architecture defines a set of components for the subscription management. The storage layer is composed of the multi-tenant database and the logical persistence separation. Different types of subscriptions are handled by a component called Subscription Type Management. As each provider can define its resources for their applications, the Restriction Management is in charge of managing these resources and linked them to a Restriction definition. The Resource Management Remote layer performs access to distributed resource managers from different and heterogeneous sources. This is an important concept of the subscription component because it has the ability to manage distributed resources in different scenarios, either for on-premise ASPs applications or SaaS solutions. As such, the subscription component uses a distributed architecture based on SOA in order to be adaptable to several scenarios. Each entity can define its own resource managers as Web services and these can be consumed for the subscription component dynamically. With this approach it is not only possible to have applications using the subscription component as well as entire platforms consuming the web services. These resources can be any type of accountable and billable resources such as persistent rows (e.g., contacts, leads, bills, surveys, etc.) or hardware (e.g., CPU cycles, bandwidth, storage, etc.). A Resource Manager interface defines a set of methods to be implemented by SaaS services. It defines methods that can be called externally due to the fact that each resource manager implementation is exposed as a Web Service in order to be consumed for the subscription service. This approach allows the dynamic integration of heterogeneous providers. Resource Manager registration is performed when a provider (e.g., a subscriber per se) defines a Restriction for each resource. Therefore, a Restriction (in the Subscription component side) will access its Resource Manager (in the application side). External providers can define their restrictions by using the Subscription Web Application front-end and this is done through the Restriction Management internal component. Internal applications of the SaaS platform are automatically analyzed by discovering their Resource Managers. The Web Services implementation in the SaaS platform is done with Apache Axis2, Spring2 and the integration library between them called WSO2 [16]. Each SaaS application implements its own resource manager, which is referenced in the *applicationContext.xml* Spring file. The subscription service implements multi-tenancy subscriptions with logical separations in its database, by using a subscriber ID field, in order to manage multiple subscribers and subscriptions. A subscriber can be any entity that has resources to bill or to consume. A subscription is a tree-relationship entity composed by two subscribers (client and provider) and a SaaS application. Therefore, we can say that "*subscriber A is subscribed to the Contact Manager SaaS application provided by subscriber B through the subscription number 1234*", as depicted in Figure 4:

 ISBN: 978-1-61208-106-9

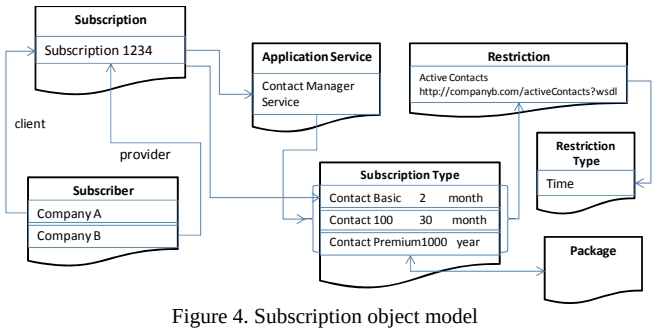

Figure 4. Subscription object model

Each provider can deploy a set of applications and their corresponding subscription types. Subscription Types are the billable plans belonging to a specific SaaS application and they are applicable to a subscription. Packages are used to combine two or more subscription types. As Figure 4 shows, the Contact Manager service defines three subscription types, depending on the contact generation rate. The subscription '1234' defines the 'Contact Basic' subscription type. As such, the 'Contact Basic' plan is going to charge 2 currency units for every active contact with a monthly fee basis. Then, an 'Active Contacts' is defined as a resource to be billed. In order to manage these resources, a subscription type has a Restriction, which is a condition to be monitored by the subscription component and it allows managing the resources' accountability. In the previous example, the 'Active Contacts' resource is defined in the Restriction table, and it defines the source of this resource (as a WSDL end-point). Subscription service implements a Resource Lookup and Invoice Generation mechanisms which use the WSDL end-points in order to gather specific information about billed resources status.

VIII. CONCLUSION

A Software-as-a-Service (SaaS) platform has been described and its implementation on open source technologies. This platform implements a set of business services and components to deploy Web applications and manage them as SaaS solutions. These SaaS applications use shared database schema in order to implement multi-tenancy mechanisms by logically separating their data for each subscriber. The authors' SaaS platform provides an easy transition from traditional ASP applications with single provider approach to more complex scenarios for the next generation of Internet-based services. As such, this platform and its business services are currently used for deploying industry-class SaaS solutions in real production environments.

ACKNOWLEDGMENT

The research presented in this document is a contribution for the "*Rapid Product Realization for Developing Markets Using Emerging Technologies*" Research Chair, ITESM, Campus Monterrey, and for the "*Design of Mechatronics Products*" and "*New Business Models for the Knowledge Economy*" Research Chairs, ITESM, Campus Ciudad de México.

REFERENCES

[1] Nurmi, D.; Wolski, R.; Grzegorczyk, C.; Obertelli, G.; Soman, S.; Youseff, L., and Zagorodnov, D. "*The Eucalyptus Open-Source Cloud-Computing System*". May 2009. CCGRID '09. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 90-100.

[2] Molina, A.; Mejía R..; Galeano N.; Najera T., and Velandia M**.** "*The HUB as an Enabling IT Strategy to Achieve Smart Organizations*". Chapter III in Integration of ICT in Smart Organizations, Istvan Mezgar (Editor), Idea Group Publishing, 2006, pp. 64-95.

[3] Concha, D.; Espadas, J.; Romero, D., and Molina, A. "*The e-HUB Evolution: From a Custom Software Architecture to a Software-as-a-Service Implementation*". 2010. Journal Of Computers In Industry. Volume 61, Issue 2, pp. 145-151.

[4] Espadas, J.; Concha, D. and Molina, A. "*Application Development over Software-as-a-Service Platforms*". 2008. The Third International Conference on Software Engineering Advances ICSEA 2008, pp. 97-104.

[5] Chou, T. "*The End of Software*". Sams Publishing, USA, 2005.

[6] Jacobs, J. "*Enterprise software as service*". July 2005. Queue, Volume 3, Issue 6, pp. 36-42.

[7] Dubey, A. McKinsey. Panel at the SIIA OnDemand Summit. San Jose, CA. November 8, 2006.

[8] Gartner Research. "*Predicts 2007: Software as a Service Provides a Viable Delivery Model*". 2006 Gartner, Inc.

[9] Natis, Y. V. "*Introducing SaaS-Enabled Application Platforms: Features, Roles and Futures*". 2007 Gartner, Inc.

[10] Pring, B.; Bona, A.; Holincheck, J.; Cantara, M. and Natis, Y. V. "*Predicts 2008: SaaS Gathers Momentum and Impact*". January 2008. Gartner Inc.

[11] Wolde, E. Research Analyst, IDC. August 2007.

[12] Aulbach, S.; Grust, T.; Jacobs, D.; Kemper, A., and Rittinger, J. "*Multi-tenant databases for software as a service: schema-mapping techniques*". June 2008. SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 1195-1206.

[13] Mietzner, R.; Metzger, A.; Leymann, F., and Pohl, K. "*Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications*". May 2009. PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25.

[14] Frederick C.; Gianpaolo C., and Roger W. (June 2006). "*Multi-Tenant Data Architecture*". MSDN Library. Microsoft Corporation. http://msdn.microsoft.com/en-us/library/aa479086.aspx. Last access at January 2009.

[15] Sedayao, J. (November, 2008). "*Implementing and operating an internet scale distributed application using service oriented architecture principles and cloud computing infrastructure*". iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, pp. 417-421.

[16] Mathew, T. (February, 2008). "*Hello World with WSO2 WSF/Spring*". WSO2 - The Developer Portal for SOA. http://wso2.org/library/3208. Last access at June 2009.

[17] Haines, S. (Sept, 2005). "*JoSQL - SQL for Java Objects*". http://www.informit.com/guides/content.aspx?g=java&seqNum=230. Last access at July 2009.

[18] Kan, W. and Yujun, Z. (2009). "*Using LINQ as an instructional bridge between object-oriented and database programming*". ICCSE '09. 4th International Conference on Computer Science & Education, 2009, pp. 1464 - 1468

 ISBN: 978-1-61208-106-9