The SPOSAD Architectural Style for Multi-tenant Software Applications

Heiko Koziolek
Industrial Software Systems
ABB Corporate Research
Ladenburg, Germany
heiko.koziolek@de.abb.com

***Keywords*-Software architecture; Software quality; Software performance; Software maintenance**

***Abstract*—A multi-tenant software application is a special type of highly scalable, hosted software, in which the application and its infrastructure are shared among multiple tenants to save development and maintenance costs. The limited understanding of the underlying architectural concepts still prevents many software architects from designing such a system. Existing documentation on multi-tenant software architectures is either technology-specific or database-centric. A more technology-independent perspective is required to enable wide-spread adoption of multi-tenant architectures. We propose the SPOSAD architectural style, which describes the components, connectors, and data elements of a multi-tenant architecture as well as constraints imposed on these elements. This paper describes the benefits of a such an architecture and the trade-offs for the related design decisions. To evaluate our proposal, we illustrate how concepts of the style help to make current Platform-as-a-Service (PaaS) environments, such as Force.com, Windows Azure, and Google App Engine scalable and customizable.**

I. Introduction

The software industry is currently adopting the Software-as-a-Service (SaaS) deployment model in many application domains [1]. SaaS applications are hosted on Internet servers by a provider instead of being downloaded and installed locally on the user's computer. SaaS applications typically provide licenses on a pay-per-use basis, instead of being bought by a user.

A special kind of SaaS offering is a multi-tenant software application [2]. It serves multiple tenants (e.g., companies or non-profit groups) from a single application instance. Furthermore, for all tenants the software runs on the same infrastructure (i.e., containers, virtual machine, operating system, hardware), runs from the same code base, and can thus be maintained centrally. The most popular example for a multi-tenant SaaS application is a customer relationship management (CRM) software by Salesforce [3].

Because of their scalability and efficiency, multi-tenant architectures are considered a critical competitive advantage over classical single-tenant architectures for large-scale systems in specific domains, such as CRM, HR, or financial applications [2]. Despite the emerging platforms for SaaS applications [1], implementing multi-tenant architectures is still not well understood and documented. It is hard for software engineers to develop such a system, because the architectural concepts, the necessary design decisions and their trade-offs require high expertise.

Existing attempts on providing a structured documentation of multi-tenancy architectural concepts either focus on restricted aspects (e.g., the database layer [4]) or depend on certain technologies (e.g., .NET [5], IBM/Java [6]). The documented architectures of existing multi-tenant systems (e.g., [3]) are difficult to transfer to other systems. A more abstract and technology-independent perspective is needed to understand the involved design decisions and architectural trade-offs.

We propose capturing the architectural concepts of a multi-tenant system as a new *architectural style* (the so-called SPOSAD style: **S**hared, **Po**lymorphic, **S**calable **A**pplication and **D**ata). Classical architectural styles (e.g., client/server, pipe-and-filter, mobile code, etc.) were documented decades ago, but are still useful to structure new software applications [7]. The SPOSAD style for multi-tenant applications as described in this paper is an extension of the multi-tier architectural style [7].

The contribution of this paper is the initial proposal of a new architectural style called SPOSAD for multi-tenant software systems. We apply software architecture research concepts and methodologies to multi-tenant software systems to help software architects understand the implications of their design decisions and inevitable trade-offs. We have enhanced the initially sketched concepts from a former paper [8] to comprise multiple architectural views of SPOSAD, detailed descriptions of the involved components, and a refined presentation of the induced architectural constraints. To initially evaluate the style, we compare its architectural concepts with the concepts underlying three Platform-as-a-Service (PaaS) environments (i.e., force.com, Windows Azure, and Google App Engine).

This paper is organized as follows: Section II briefly recalls some background on architectural styles and describes two recent styles related to SPOSAD. Section III then describes the SPOSAD style with the desired architectural properties, its architectural elements, its induced constraints on these elements, and its trade-offs. Section IV provides an initial evaluation of the style via a relation to current PaaS environments. Finally, Section V discusses related work.

II. Architectural Styles

According to Fielding [9], an architectural style is a coordinated set of architectural constraints that restricts the roles of the architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. Compared to the more descriptive design patterns that summarize proven solutions, architectural styles have a more prescriptive character and limit the design space.

Basic architectural styles are for example client/server, n-tier, pipe-and-filter, and code on demand. More complex styles, which build on and extend the basic styles, are model-view-controller for GUIs, REST for the WWW [9], and SPIAR for AJAX applications [10].

REST and SPIAR are architectural styles related to multi-tenant architectures. REST induces a constrained client/server architecture with focus on the communicated data elements. The style prescribes the use of resources (i.e., the target of hyperlinks), resource identifiers (e.g., URLs), representations (e.g., HTML documents, JPEG images), and meta-data. Two constraints for the architecture are a synchronous request/response communication between client and server as well as stateless and context-free interaction for scalability.

SPIAR targets client / server architectures with rich user interfaces and was deduced from AJAX applications. The style constraints the architecture by prescribing asynchronous interaction between client and server, delta-communication (i.e., only state-changes are transferred to reduce network traffic), and component-based user interface for more interactivity.

Both the REST and SPIAR style (if stateless) might be used in a multi-tenant architecture. However, they are not sufficient to describe such architectures. Multi-tenancy puts additional constraints on the code to be used at the application tier and the data elements held in the data tier as described in the next section.

To describe the architectural concepts in this paper, we use the terminology of Perry and Wolf [11]. They define an architecture as a configuration of architectural elements - processing (i.e., components), connectors, and data - constrained in their relationships in order to achieve a desired set of architectural properties.

III. The SPOSAD style

This section first describes the architectural properties targeted by the SPOSAD style (Section III-A), before describing its architectural elements in detail (Section III-B). Then, the architectural constraints imposed by SPOSAD are elaborated (Section III-C) and architectural trade-offs are discussed (Section III-D). The section concludes with a discussion of the style, a short comparison to related approaches, and open issues (Section III-E).

A. Architectural Properties

The following architectural properties mainly focus on the extra-functional properties to be achieved by the style. They describe the goals of a multi-tenant software architecture and can also be viewed as typical requirements for such a system.

Resource Sharing: One central aim of a multi-tenant architecture is cost reduction by sharing resources among a large number of tenants. Resources do not only include hardware devices, such as CPUs, memory, and hard disks, but also software resources, such as application servers, databases, and operating systems. It is the aim to leverage economics of scale and avoid unnecessary overheads for hardware and resources. The total costs of ownership (TCO) shall be reduced for the tenants, because they face lower maintenance costs for patching as well as for up- and down-scaling.

Scalability / Elasticity: A multi-tenant system shall serve a large number of tenants with a potentially also large number of clients. Such a system should be able to handle growing amounts of workload without noticeable increasing response times or decreased reliability. Also, reduced amounts of workload shall be handled efficiently and seamlessly.. Due to the inherent limits of scaling up (i.e., adding resources to a single node), the ability to seamlessly scale out (i.e., adding more nodes) is a typical architectural property of a multi-tenant system.

Maintainability: In contrast to many hosted, single-tenant software applications, a multi-tenant architecture relies on a shared code basis for all tenants. The developers can fix bugs centrally. Feature updates are available to all tenants. Thus, the costs of maintaining a single system should be dramatically lower than for multiple systems. Besides the code, also the database administration shall be shared among tenants to reduce maintenance costs for databases.

Customizability: While a single code base limits a multi-tenant system for high customization by individual tenants, a limited degree of user extensions and tenant-specific adaptations shall still be possible. This is simply a functional requirement from many tenants, who want to use adaptations of the software for customer-friendliness and competitive advantages. A well-designed multi-tenant architecture effectively trades resource sharing off against user customizability.

Usability: Besides changing the business logic and the data of an application, also the user interface shall be configurable through tenant-specific customizations. It allows different tenants to create their own branding for an application.

B. Architectural Elements

This section describes the architectural elements of the SPOSAD style, namely its components, its connectors, and its data elements according to the template by Perry and Wolf [11] for the description of architectural styles. Multiple views of the SPOSAD style (Fig. 1- 4) illustrate the relationships between its elements.

1) Components: The components of the SPOSAD style are arranged as in the classical multi-tier style into several tiers (cf. Fig. 1). This improves the separation of concerns and enables updating certain tiers (e.g., the database tier) during system evolution. Users access the system through the client tier and access the application and database tier through the presentation tier, which is responsible for providing the user interface (e.g., web pages). For business transactions, all computations are made in the application tier, which retrieves and persists the involved data in the database tier.

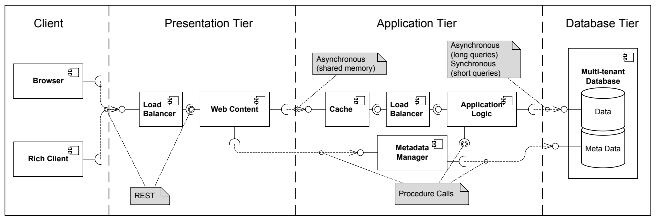


Figure 1: Functional view of the SPOSAD style comprising components and connectors

The following components (Fig. 1) are typical (but not sufficient) for a multi-tenant system:

Browser/Rich Client: Users access the system via a web browser as in typical web applications or a rich client application (using HTML/REST).

Web Content: The Web Content component is responsible for serving HTTP requests and providing static (via HTML) or dynamic pages (via CGI, servlets, JSPs, ASP.NET, etc.). This component typically spawns several threads (cf. Fig. 2) to handle individual client requests. In the SPOSAD style, the threads of this component are constraint to be stateless to keep them independent from specific clients. Thus, immediately after completing a single request by a user, each thread can serve another user without waiting for input from the first user.

For the multi-tenant system conforming to SPOSAD, the viewable content shown by the Web Content component must be customizable, which is not provided by regular web applications. Each tenant may provide a unique arrangement of the user interface, branding, and specialized forms for tenant-specific data. Therefore, the Web Content component may access the meta-data manager within the application tier to provide tenant-specific customizations for the user interface.

Load Balancer: Requests directed to the Web Content component and Application Logic component are routed through one or more load balancers, which ensure a uniform utilization of the respective threads. Different balancing strategies might be useful depending on the expected request profile.

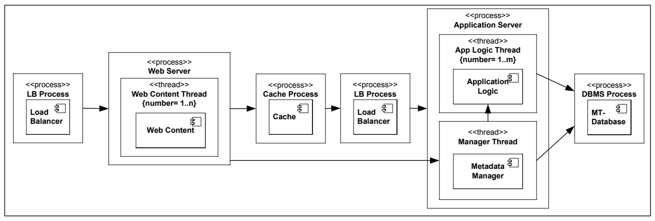


Figure 2: Concurrency view of the SPOSAD style with processes and threading

The load balancers ensure horizontal scaling, so that the application can be adapted to an increased number of requests by adding more machines instead of using more powerful machines. Load balancers in multi-tenant systems typically also support auto-scaling, which includes shutting down unnecessary machines if the request load decreases (”elasticity”). This can lower the overall power consumption of a data center and lead to substantial financial savings.

Application Logic: The Application Logic component executes tenant-specific business logic and queries the database for necessary data. Unlike in a single-tenant system, where tenant-specific implementations of the application logic are created and maintained, in a SPOSAD-constrained multi-tenant system the Application Logic component is based on the same source code for all tenants. Thus, maintaining the common code benefits all tenants and patches can be applied centrally.

Unlike in the classical multi-tier architecture, the Application Logic might be adapted by the meta-data manager for tenant-specific workflows or computations and thus exhibits a so-called ”polymorphic” behaviour. Constrained by SPOSAD, the threads of the application logic reside in a single container on the same machine to increase resource sharing, but there might be multiple machines hosting instances of the component (Fig. 3). Communication with other components can be handled by middleware.

In regular web applications, the application logic may carry client-specific state. However in the SPOSAD style, the Application Logic must be stateless to ensure scalability. It allows I/O operations to be carried out asynchronously. This enables horizontal scaling. User-specific state must be stored either on the client side or in the database, which can increase the complexity of the application.

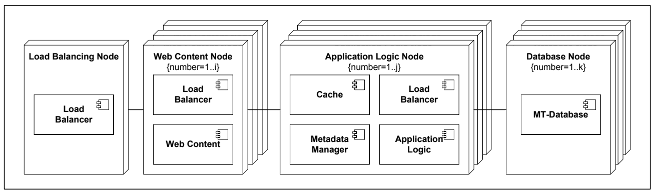


Figure 3: Deployment view of the SPOSAD style illustrating one possible component allocation

Meta-data Manager: The Meta-data Manager is responsible for customization of the application logic threads and the web content threads. It retreives tenant-specific meta-data from the database and adapts the application accordingly. Different implementations are possible: the Meta-data manager can direct meta-data to the application logic and let the application adapt itself or the Meta-data Manager can generate tenant-specific code on-the-fly, which can then serve as the application logic.

There may be different scopes of customization per tenant, so that, for example, different business units of a company can be handled differently. According to SPOSAD, the multi-tenant system must allow users to customize the application logic and data model and must provide respective functionalities (e.g., wizards) for this.

Multi-tenant Database: The multi-tenant database stores business data as well as meta-data persistently. To maximize resource sharing, a single database should host the data of all tenants. It avoids overheads for memory consumption and administration.

A conventional, off-the-shelf database can be used for a multi-tenant system. However, the data needs a special arrangement to enable multi-tenancy. Different options for the data and schema layout for multi-tenant applications are discussed Section III-B3.

Hosting a large amount of tenants in the same database results in the need to partition the database and to store the data into multiple physical nodes. Special security mechanisms, such as row level access control, are required to keep the tenant data isolated from each other. Backup procedures can be implemented efficiently by creating table spaces from shared tables for each tenant and making incremental backups [2], [4].

2) Connectors: The communication between the client and the presentation tier follows a RESTful style. Clients request services from the Web Content component synchronously.

Between the presentation tier and the application tier SPOSAD requires the communication to be asynchronous, so that application threads do not have to wait for user inputs.

The interaction of other components with the meta-data manager is handled through procedure calls. Between the application tier and the database tier, the communication can be synchronous for short queries and asynchronous for long running queries. With the asynchronous communication, waiting for the query results from the multi-tenant database can be avoided, while the short synchronous queries can be handled immediately.

The connections might involve additional resolvers (e.g., DNS lookup) or tunnels (SOCKS, SSL after HTTP) as additional, optional connectors. Because the component topology follows the pipe-and-filter style, the communication can be flexibly extended.

3) Data Elements: Requests exchanged by the components, as well as the data stored in the database tier, represent the most important data elements in the multi-tenant system. Besides the client id, each client request must include also the tenant id, so that tenant-specific customizations and security mechanisms are enabled.

In the data tier, as many resources as possible should be shared to decrease operational expenses. For example, when hosting data for a large number of tenants, the memory and processing overheads for keeping their data in separate databases or tables should be avoided. The multi-tenant database stores at least tenant-specific customer data (e.g. accounting information) and tenant-specific meta-data (e.g., customized workflows).

There are several trade-offs when implementing a form of sharing in the data tier [5]. The simplest solution is using a *separate database* per tenant on the same physical nodes, which is straight forward to implement and favorable for security purposes. However, the sharing of resources is limited and costs for hardware, backups, and administration can be high.

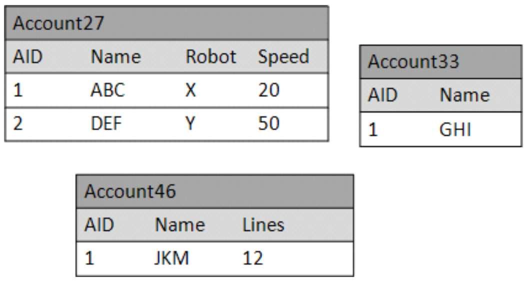
A shared database with *per-tenant schemas* reduces memory overheads, hardware costs, and maintenances efforts, because only a single database needs to be managed. Drawbacks are potential security issues and complicated backup procedures, because it is then desired to keep tenant-specific backups. Using a shared databased with a *single schema* for all clients results in even lower operational expenses because of the reduced memory overheads and low administration costs. However, ensuring security is even more complicated and special mechanisms for query optimization and partitioning are required to manage the potentially large tables.

Different schema mapping techniques (i.e., layouts) for multi-tenant databases hosting the data of multiple clients are shown in an information view of the SPOSAD style in Fig. 4 (see also Aulbach et al. [4]). The same data is shown for three different layouts.

In a *private table* layout, each tenant gets its own tables with potentially tenant-specific columns and data types. In Fig. 4a, the tenant ids are 27, 33, and 46 and each tenant has its own table with proprietary columns. Because of the memory overheads for storing tables, this layout is unfavorable when hosting a large number of tenants (e.g., several thousands).

In an *extension table* layout, all common tenant data is stored in a central table, while only tenant-specific extensions are realised by additional tables (Fig. 4b). If many tenants customize the schema, this again results in a high number of required tables and additionally requires joins when accessing the data.

The *universal table* layout stores all tenant data in a single table. The table contains multiple columns with a generic data type (e.g. varchar), which can be filled with tenant-specific data (Fig. 4c). No expensive joins are needed to

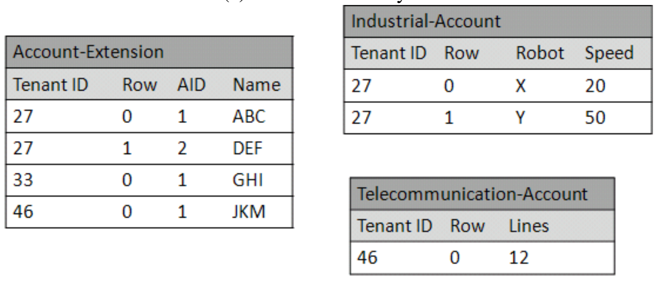


Account27			
AID	Name	Robot	Speed
1	ABC	X	20
2	DEF	Y	50

Account33	
AID	Name
1	GHI

Account46		
AID	Name	Lines
1	JKM	12

(a) Private Table Layout

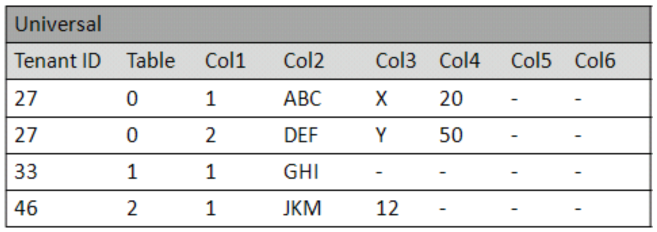


Account-Extension			
Tenant ID	Row	AID	Name
27	0	1	ABC
27	1	2	DEF
33	0	1	GHI
46	0	1	JKM

Industrial-Account			
Tenant ID	Row	Robot	Speed
27	0	X	20
27	1	Y	50

Telecommunication-Account		
Tenant ID	Row	Lines
46	0	12

(b) Extension Table Layout



Universal							
Tenant ID	Table	Col1	Col2	Col3	Col4	Col5	Col6
27	0	1	ABC	X	20	-	-
27	0	2	DEF	Y	50	-	-
33	1	1	GHI	-	-	-	-
46	2	1	JKM	12	-	-	-

(c) Universal Table Layout

Figure 4: Information view of the SPOSAD style showing data layouts in the multi-tenant database

reconstruct the logical schema, but the table might store many null values and type conversion might be required.

C. Architectural Constraints

The following constraints restrict the architectural elements introduced before and help to avoid design decisions with a negative impact on the desired architectural properties.

Single code base: For the web content and application logic components, there must be a single code base shared among all tenants. Tenant-specific extensions to the code are not allowed. Tenant-specific workflows or data must be introduced using meta-data. The single code base including a single configuration management and bug tracking system improves the maintainability of the system. Patches only need to be applied once to the shared code and are then available for all tenants.

Shared resources in the database tier: Hosting a dedicated database server per tenant with proprietary data schemas does not comply to the SPOSAD style. Some sharing of data resources as in Fig 4 is required, which also differentiates the SPOSAD style from the n-tier style. Sharing avoids memory overheads and wasted resources for hosting underused database servers. It enables large scalability of the system, because DBMS-specific facilities for partitioning and distributing the data can be used.

Customizability via meta-data: The application logic threads, the web content, and the data schema must be customisable using meta-data. While resource sharing is the most desirable property of the SPOSAD style, tenant-specific customizations are a necessity from a business perspective, as tenants will not accept standard solutions in many cases.

Stateless application tier: The application tier must not hold client-specific state, such as transactional data or inputs of user forms. This constraint allows for efficient usage of the processing resources, as the application threads do not have to wait for user inputs of a specific client, but can process requests by other users in the mean time.

Asynchronous interaction with the application tier: The communication between the presentation tier and the application tier, as well as between the application tier and the database tier must be asynchronous where possible to avoid waiting delays.

Besides the listed constraints, the SPOSAD styles inherits all constraints of the multi-tier style [7] (e.g., prohibiting clients from directly accessing the data tier).

D. Architectural Trade-offs

On the one hand, applying the SPOSAD style results in several trade-offs, which should be carefully considered before deciding to implement a multi-tenant system. On the other hand, the SPOSAD style still leaves several design options open, so that architects have to weigh different requirements and determine their trade-offs themselves.

Complexity vs. Time-to-Market: A multi-tenant, hosted application running a single instance for all tenants is significantly more *complex* to implement than a multi-instance application. Mechanisms for keeping the application customizable have to be found and the database has to be specially prepared. Therefore, a system according to the SPOSAD style requires higher developer skills.

Security/Availability vs. Resource Sharing: Because multiple clients of multiple tenants share the same infrastructure in a multi-tenant system, the architects needs to ensure that the clients work in *isolation* and do not affect each other. Hosting business-critical data of multiple tenants in the same infrastructure or even the same database table requires special measures for keeping the data logically isolated (e.g., using encryption). It has to be ensured that the application threads of one tenant do not interfere with application threads by other tenants and crash the underlying VM or decrease the overall performance. Reliability measures might include application thread replication and the isolation of performance-intensive application tasks onto individual VMs.

Customizability vs. Maintainability The architect has to define the degree of *customization* that the application should support. More customizability implies more complicated development and makes the use of shared resources more difficult. Thus, highly customizable applications are not well suited for a multi-tenant architecture. The architect should identify an adequate number of variation points in the software.

E. Discussion

It is helpful to delimit multi-tenant architectures from single-tenant, n-tier architectures to make their special features better comprehensible. For example, an application such as Hotmail hosts the data of multiple tenants in the same infrastructure, but does not allow for tenant-specific customizations using meta-data. The SaaS application by SAP for small companies called BusinessByDesign hosts the clients of each tenant on a dedicated physical machine[1]. Thus, it can be considered a single-tenant solution.

Infrastructure-as-a-Service (IaaS) solutions, such as Amazon EC2[2] offer virtualized servers, but no out-of-the box support for application sharing. Hosted software on Amazon EC2 is usually single-tenant, as dedicated virtual servers and databases are set-up per tenant. Compared to a multi-tenant application such an approach might lead to a waste of resources, such as memory, disk space, and CPU power, because underutilized servers cannot handle the requests of other tenants. However, such a solution can be easier to implement when dealing with legacy software.

Some issues are still open to complete the description of the SPOSAD style and can be considered future work. The role of application servers and middleware needs to be discussed with greater detail. Special requirements for the database in terms of query optimizations and backup procedures need to be documented. Furthermore, facilities for usage-based metering and payment of service per tenant should be addressed with a dedicated component.

IV. Evaluation

The evaluation of an architectural style is inherently difficult [9], [10]. Its success depends on whether systems complying to the style in fact show the promised architectural properties, such as scalability or usability. Frameworks need to be created to support developing according to the style. Furthermore, the efforts for an ad-hoc development of a multi-tenant system have to be compared with the efforts for a development based on the style. This requires expensive, repeated controlled experiments across multiple development projects, where the interfering variables are be hard to control.

To provide an initial form of evaluation in the scope of this paper, we have investigated currently emerging PaaS environments for their architectural properties. We analyse Force.com in Section IV-A, Windows Azure in Section IV-B, and Google App Engine in Section IV-C.

These environments can ease the implementation of multi-tenant systems and can be considered as frameworks, because their infrastructure is already prepared for high scalability and resource sharing following some principles also present in the SPOSAD style. However, they leave many design options open for developers and do not enforce all architectural constraints bundled in the SPOSAD style.

A. Force.com

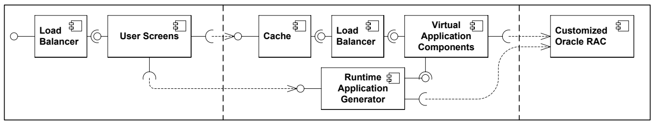


Figure 5: Simplified view on the Force.com architecture

With the force.com platform developers may build applications on top of the salesforce.com infrastructure. On a high level of abstraction, the platform is built according to an n-tier architecture [3] comprising a presentation tier, an application tier, and a data tier (Fig. 5).

Clients access the application tier of force.com according to the REST style. Each tenant is served by application instances originating from the same code base. Salesforce manages updates of this code base centrally. Tenants can customize the application user interface (forms), business logic (workflows), and data (customized tables) by specifying meta-data stored in the so-called Universal Data Dictionary (UDD). A runtime application generator creates tenant-specific application logic from this meta-data. Thus, the application is considered 'polymorphic', as it appears and behaves differently for the clients of each tenant.

Through the application tier, all tenants access the same logical database in the data tier, which is a customized version of an Oracle database. All tenant data is stored in a single table, which can be partitioned among multiple machines. Besides a tenant id column, the table contains 500 customizable columns (varchar datatype) for storing arbitrary data (i.e., a universal table layout, cf. Fig. 4).

Salesforce has claimed that it hosts more than 50000 tenants and 1.5 million subscribers on only 1000 servers [12], thus demonstrating the scalability of the architecture. Furthermore, patches are applied weekly to the single code base and thus rolled out to all customers, thereby decreasing maintenance costs.

B. Windows Azure

The Windows Azure platform by Microsoft allows deploying and running ASP.NET and WCF applications in Microsoft data centers [13]. The data centers run the Windows Azure Hypervisor and modified versions of Windows Server

[1]goo.gl/CQWD
[2]aws.amazon.com/ec2/

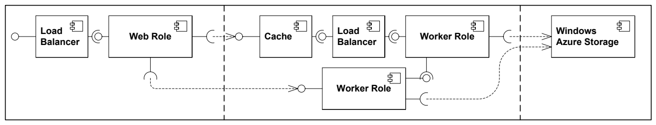


Figure 6: Simplified view on the Windows Azure architecture

2008 on a large number of virtual machines. The platform follows a three-tier structure (Fig. 6).

Clients, such as browsers or web services, access the application tier using REST or SOAP. In the application tier, applications with a UI are implemented as so-called 'web-roles', while background applications are implemented as so-called 'worker-roles'. Web-roles and worker-roles may interact asynchronously using queues. Ideally, they are stateless and may be run in a configurable number of instances. Load balancers can distribute requests among those instances. Tenants can implement UI customizations using MS Silverlight and business logic customizations using Windows Workflows.

Web/Worker-roles can either access the non-relational, horizontal scalable Windows Azure storage or slightly customized versions of the MS SQL server (SQL Azure). The Windows Azure storage features blobs, non-relational tables similar to the universal table layout, and queues for data persistency. Queues handle interaction between web- and worker roles by storing data portions.

Case studies reported on the Azure website[3] indicate that the platform can support highly scalable applications. For example, the online auction company Adslot successfully ran simulations with more than 100 stateless worker roles and more than 4000 users accessing the system without experiencing performance problems. A single code basis is not enforced by Windows Azure and thus needs to be adhered by the SaaS providers themselves.

C. Google App Engine

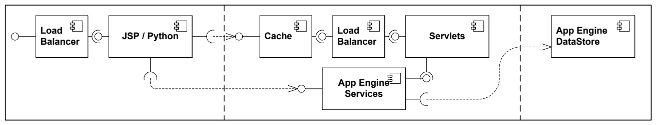


Figure 7: Simplified view on the Google App Engine architecture

Google's App Engine (GAE) [14] enables developers to run Java or Python applications in Google's data centers. Several web frameworks, such as Django, Cherrypy, or pylons run on GAE and assist developers in implementing their applications. Again, the architecture features a presentation, application, and data tier (Fig. 7).

[3]www.microsoft.com/windowsazure/

Clients access the application tier of GAE using REST. Besides responding to web requests, GAE also allows to run so-called 'scheduled tasks' as possibly periodic background tasks. Web applications and background tasks might interact asynchronously using queues. GAE features built-in auto-scaling, load balancing, and fail-over mechanisms between identical implementation instances in the application tier. GAE also allows the management of multiple, tenant-specific namespaces to ease the implementation of multitenancy applications.

Applications may store data in a non-relational structure called Google Big Table, which shall be able to handle large-scale applications and store petabytes of data. The GAE storage features a proprietary query engine with the Google Query Language that allows transactions. The Big Table does not have a schema, the structure of the contained data entities must be provided and enforced by the application code.

GAE has proven to be scalable during a town hall meeting of US president Obama in 2009 [15]. Using Google Moderator running on GAE, more than 100,000 questions were asked by more than 90,000 users within 1.5 hours, which did not result in significant performance degradation.

V. Related Work

There is still limited scientific research on multi-tenant architectures. The underlying concepts evolved in SaaS companies and have been used as a competitive advantage. They have hardly been analysed systematically by researchers.

Existing documentations of multi-tenant architectures are often technology-specific. Chong and Carraro from Microsoft discuss the business rationale of SaaS applications and describe their high level architectural concepts [2], [5]. However, they do not describe a reusable architectural style.

The description by Goldszmidt et al. [6] highlights some architectural concepts of multi-tenant software applications, but is centered around Java and IBM products, such as the Websphere application server or DB2. Weissman [3] provides an overview of the force.com architecture. His description is not easily transferable to other multi-tenant architectures.

Aulbach et al. [4] provide a database centric view on multi-tenant architectures. They evaluate the performance properties of different flexible schemas for multi-tenant applications and propose a new, more efficient schema. Their analysis lacks an evaluation of the scalable storage solutions of current PaaS environments. Furthermore, they neglect the application layer of multi-tenant applications.

Wang et al. [16] proposed a framework for implementing multi-tenant applications. They describe patterns for security, performance, and administrations isolation in such architectures and sketch customization concepts. However, they neglect the application tier in their investigation.

Kwok et al. [17] deal with capacity planning in multi-tenant applications and propose a method for determining the optimal allocation of application threads to physical nodes. Mietzner et al. [18] extend the service component architecture (SCA) to be able to describe multi-tenant applications.

VI. Conclusions

We have initially proposed a new architectural style called SPOSAD for multi-tenant software applications. This paper has defined architectural constraints, discussed trade-offs to consider by the architect. We have analyzed our proposal through a comparison with current PaaS environments.

The contribution of this paper is the application of software architecture research concepts and methodologies, such as architectural constraints and style descriptions, on the current practices of efficiently scalable SaaS applications. Ultimately, it shall help software architects to make more informed design decisions and to implement multi-tenant systems more efficiently.

The style description in this paper bears many directions for future work. The role of application containers and virtualization techniques needs to be discussed with greater detail. Patterns for security, scalability, and reliability could augment the style description.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," UC Berkeley, Tech. Rep. 2009-28, 2009.

[2] F. Chong and G. Carraro, "Architecture Strategies for Catching the Long Tail," Microsoft Corporation, http://msdn.microsoft.com/en-us/library/aa479069.aspx, Tech. Rep., April 2006, last visited 2010-02-19.

[3] C. D. Weissman and S. Bobrowski, "The Design of the Force.com Multitenant Internet Application Development Platform," in *Proc. 35th SIGMOD International Conference on Management of Data (SIGMOD '09)*. New York, NY, USA: ACM, 2009, pp. 889–896.

[4] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-tenant databases for software as a service: schema-mapping techniques," in *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*. New York, NY, USA: ACM, 2008, pp. 1195–1206.

[5] F. Chong and G. Carraro, "Multi-tenant Data Architecture," Microsoft Cooperation, http://msdn.microsoft.com/en-us/library/aa479086.aspx, Tech. Rep., June 2006, last visited 2010-02-19. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa479086.aspx

[6] G. Goldszmidt and I. Poddar, "Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware," April 2008, last visited 2010-02-19. [Online]. Available: http://bit.ly/ahVZdp

[7] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[8] H. Koziolek, "Towards an Architectural Style for Multi-tenant Software Applications," in *Proc. Software Engineering 2010 (SE'10), Fachtagung des GI-Fachbereichs Softwaretechnik*, ser. LNI, vol. To appear. GI, February 2010.

[9] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, 2002.

[10] A. Mesbah and A. van Deursen, "A component- and push-based architectural style for AJAX applications," *J. Syst. Softw.*, vol. 81, no. 12, pp. 2194–2209, 2008.

[11] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[12] E. Schonfeld, "The efficient cloud: All of salesforce runs on only 1,000 servers," http://tcrn.ch/b35s9T, March 2009, last visited 2010-02-19.

[13] D. Chappell, "Introducing the Windows Azure Platform," DavidChappell & Associates, http://go.microsoft.com/fwlink/?LinkId=158011, Tech. Rep., August 2009, last visited 2010-02-19.

[14] Google, "App engine," http://appengine.google.com, last visited 2010-02-19.

[15] T. Janisch, "Obamas online town hall provides better data than answers," http://bit.ly/dwEXJd, May 2009, last visited 2010-02-19.

[16] Z. H. Wang, C. J. Guo, B. Gao, W. Sun, Z. Zhang, and W. H. An, "A Study and Performance Evaluation of the Multi-Tenant Data Tier Design Patterns for Service Oriented Computing," in *Proc. Int. Conf. on E-Business Enigneering (ICEBE'08)*. IEEE, 2008, pp. 94–101.

[17] T. Kwok and A. Mohindra, "Resource calculations with constraints, and placement of tenants and instances for multi-tenant saas applications," in *Proc. 6th Int. Conf. on Service-Oriented Computing (ICSOC'08)*, 2008, pp. 633–648.

[18] R. Mietzner, F. Leymann, and M. P. Papazoglou, "Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns," in *Proc. 3rd Int. Conf. on Internet and Web Applications and Services (ICIW'08)*. IEEE Computer Society, 2008, pp. 156–161.