ARCHITECTURAL CONCERNS IN MULTI-TENANT SaaS APPLICATIONS

Rouven Krebs[1], Christof Momm[1] and Samuel Kounev[2]
[1]*SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany*
[2]*Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany*

Keywords: Multi-tenancy, SaaS, Platform, Resource Sharing, Affinity, Performance Isolation.

Abstract: Multi-tenant applications serve different customers with one application instance. This architectural style leverages sharing and economies of scale to provide cost efficient hosting. As multi-tenancy is a new concept, a common definition of the word and related concepts is not yet established and the architectural concerns are not fully understood. This paper provides an overview of important architectural concerns and there mutual influences. Beside that, it defines multi-tenancy and differentiates it from several related concepts.

1 INTRODUCTION

Software-as-a-Service (SaaS) is a cloud service offering comprising a ready to run, hosted application like a CRM[1] or a Web mailer[2]. To reduce the total costs of ownership (TCO), some of these SaaS offerings allocate users of different customers to the same application instance. If the users of one customer represent a closed group, which is usually charged and handled as a single entity, they are referred as a tenant. Applications designed to serve multiple tenants with a single runtime instance are referred as multi-tenant applications (MTA).

Sharing resources in general yields cost benefits in cloud environments (Schuller, 2009). This is most efficient on the level of application instances (Momm and Krebs, 2011), since in this case the general overhead is minimal and it is possible to perform a very fine grained workload and resource management (e.g. requests, threads).

To still meet customers' expectations regarding the service levels, systems have to ensure a functional and non-functional isolation of the tenants, which is a major challenge, especially when the application instance itself is shared.

However, from a research point of view multi-tenancy is a field that still introduces many undefined concepts and acronyms. This causes confusion, not only because of different definitions of multi-tenancy but also because of various biased assumptions about its implementation.

The major contribution of this paper is a discussion of software architecturel concerns for implementing multi-tenant applications by pointing out characterizing features and differentiating aspects from other related concepts such as system virtualization.

The remainder of the paper is structured as follows. Section 2 provides a definition of multi-tenancy, including a discussion of related concepts. Section 3 points out the major architectural concerns when designing MTAs and section 4 discusses the relationships between these concerns. Section 5 presents the related work, while section 6 concludes this paper.

2 SHARING CONCEPTS AND MULTI-TENANCY

This section introduces different approaches for sharing resources in cloud environments and concise definitions for the terms *tenant* and *multi-tenancy*.

The following section briefly explains different commonly used sharing mechanisms based on (Osipov et al., 2009) (Guo et al., 2007) and discuss their relationship to our notion of multi-tenancy.

2.1 Different Layers

When a *single code base* is shared between different

[1]e.g. http://www.salesforce.com
[2]e.g. http://www.googlemail.com

Krebs R., Momm C. and Kounev S..
ARCHITECTURAL CONCERNS IN MULTI-TENANT SaaS APPLICATIONS.
DOI: 10.5220/0003957604260431
In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science* (CLOSER-2012), pages 426-431
ISBN: 978-989-8565-05-1

customers/tenants it comes up with some requirements for the code. If one single code base is used the application has to be widely configurable to be adapted for customer specific needs. Sharing the code base yields reuse and is omnipresent. Nevertheless, a single code base is not sufficient enough to reduce the operational costs. Developing widely configurable software instead of customer specific branches is a question related to product line engineering and not specific for MTAs.

Sharing a data center is the lowest level of resource sharing one could imagine. Reusing the facilities environment like air conditioner or network infrastructure is the simplest way of decreasing costs. Application Service Providers already have adopted this concepts for years. However, sharing the data center only has a very limited cost saving potential, e.g. workload fluctuations of different customers can't be considered for resource optimization.

Virtualization provides an easy way for sharing a single server. Running a separate instance of the application within one VM for each customer is a first step towards efficient operation and probably todays most widely adopted sharing approach. In opposite to a shared data center, virtualization allows leveraging workload fluctuations by overcommitting the servers, while allowing a good isolation. However, the overhead of this solution per customer is still quite high. Virtualization is a well established field of research with challenges and goals on a hardware related level and should not be referred to multi-tenancy which is a concept on the applications level.

Another approach *shares the middleware*. In this scenario the middleware becomes shared by several application instances. This means, that the implementation of the application itself has low or no overhead regarding multi-tenancy. The disadvantages are the challenges regarding isolation of tenants and the overhead due to separated application instances. Hosting different applications within one middleware is not different than hosting the same application multiple times, which is a topic that is already discussed extensively in the literature.

A *Multi-tenant Application (MTA)* shares one application instance among different customers to reduce overhead the most. Handling different tenants within one application instance requires several modifications as every tenant needs its own view. MTAs started to become of interest as SaaS technologies arose and thus multi-tenancy is still not clearly defined. Therefore, we provide a definition of the terms tenant and multi-tenancy in the next section.

2.2 Definition of Multi-tenancy

A *tenant* is a group of users sharing the same view on an application they use. This view includes the data they access, the configuration, the user management, particular functionality and related non-functional properties. Usually the groups are members of different legal entities. This comes with restrictions (e.g. data security and privacy).

Multi-tenancy is an approach to share an application instance between multiple tenants by providing every tenant a dedicated "share" of the instance, which is isolated from other shares with regard to performance and data privacy. A commonly used analogy for explanation is a living complex where different parties share some of their resources like heating to reduce costs, but also love to enjoy their privacy and therefore demand a certain level of isolation (in particular when it comes to noise).

Besides multi-tenancy there is also the notion of *tenant space*. A tenant space refers to the situation where customers rent a predefined space of resources in which they can run different application instances. One example is a IaaS offering where a customer buys resources in which he installs the applications of his choice.

Scenarios where multiple applications run in one instance of the same runtime environment, sometimes are also referred to as multi-tenancy, in particular from the perspective of PaaS providers, because their customers deploy several applications. This concept we call *multiple application deployment* and not multi-tenancy, because the entities to be separated have different characteristics and the challenges in this scenario are of different nature. Thus, multi-application deployment should not be equated to multi-tenancy.

Data center-, virtualization- and middleware sharing are sometimes seen as one approach to achieve a multi-tenant like behavior for the tenant. All these approaches base on one application instance for each tenant and some references (e.g. (Calvin and Friedl, 2009)) call them *multi-instance* solutions. Consequently, they lack in sharing resources and in efficiency. The development of such applications cannot be considered to be something special. As all of these concepts for sharing resources are covered by a separate scientific field with its own challenges we propose to clearly distinguish all the mentioned concepts for the sake of differentiation.

3 HIGH LEVEL DESIGN CONCERNS

When developing an MTA one has to tackle multiple challenges (Bezemer and Zaidman, 2010) and has to find a balance between several architectural trade-off decisions (Koziolek, 2011). In the following we present different high level design concerns influencing the architecture. First, we discuss affinity and persistence concerns, which are usually transparent to the tenants. Second, we focus on three concerns that might be key differentiators for competitors: performance-isolation, service-differentiation and customizability. For each design concern we provide examples from real-life applications.

3.1 Affinity

One could easily think of applications where hundreds or even thousands of application instances serve several tens of thousands of tenants (Schonfeld, 2009). In this situation, the way the users of one tenant are distributed becomes a significant issue in the design of a MTA. Affinity defines how the requests of different users of a tenant are bound to processing nodes. Contrary to traditional request/response based systems the method is based on tenant specific attributes for the routing of requests and not user specific ones. In the following, we introduce the different types of tenant affinity and explain reasons for using them.

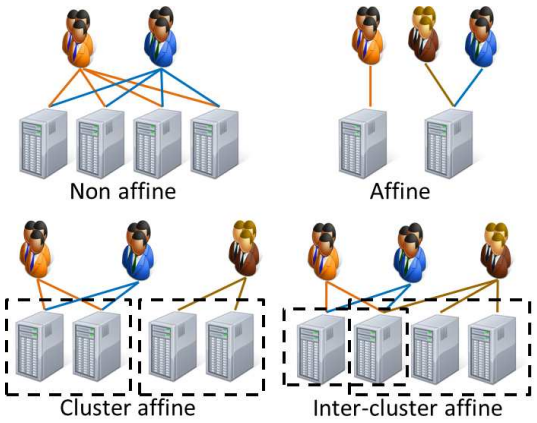

Figure 1: Different Affinities in a Multi-tenant Landscape.

In a tenant *Non-affine* application a number of application instances exist and every incoming request could be handled by any of the instances without attention to the tenant from which the request origins.

In *Server-affine* cases the requests from one tenant must be handled by the same application instance. This means, that users from one tenant cannot be distributed among several application instances but one instance can handle multiple tenants. A possible reasons for such behavior is a tenant context whereby sharing among instances is not feasible. Furthermore, a weak affinity, artificially introduced, might increase the cache hit rate.

In *Cluster-affine* situations the users of one tenant could be served by a fixed subgroup of all application instances, whereby one subgroup serves several tenants. This scenario also implies that each application instance is only part of exactly one subgroup. Keeping a tenants context, state or locking in sync among several instances or a long logical network distance could influence the performance negative and restrict the distribution.

Inter-cluster-affine is the same as the cluster affinity, but one application instance could be part of several groups. For the tenant specific context, state and locking the same arguments as for the Cluster-affine behavior arise. Legal restrictions are other aspects. For example, a server which is located in Germany is also located in the EU and thus part of two groups.

Examples. The SAP Business ByDesign solution was developed with an affine behavior in mind due to a high amount of caching. Other applications like those based on the Google's App Engine base on low tenant specific context. This means, that one tenant might be distributed over several instances of the application.

3.2 Persistence Design

Multi-tenant database designs are widely discussed (Wang et al., 2008) (Chong et al., 2006). In the following we give a short summary of the different approaches. In a *dedicated database* system, every tenant uses its own completely separated database. A *dedicated table/schema* approach shares one database which contains a separate table or schema for each tenant. In such a scenario one achieves at least a partial sharing. The *shared table/schema* approach shares the same tables and schemas, a differentiation of the data is usually provided by adding a *tenantId* column.

Examples. SAP's systems use relational data bases with a *tenentId* column. In contrast, Calvin (Calvin and Friedl, 2009) describes a solution using a separate database within Windows Azure[3].

[3]http://www.windowsazure.com

3.3 Performance Isolation

The lack of performance guarantees is one of the major obstacles for potential cloud users (bitcurrent, 2011). Performance related issues are often caused by a minority of tenants with a high workload. In the following we define performance-isolated, -weak isolated and unisolated systems with respect to performance measured and the quota each tenant has. In this case quota means the workload a tenant is allowed to produce.

Performance-isolation exists if for tenants working within their quota the SLAs are fulfilled, even if other tenants exceed their quotas. If this is met within all conditions a system is performance-isolated. An increased response time R_A for tenant A because of a high workload for tenant B is acceptable if R_A is still within the SLAs. A related concept is resource isolation, it isolates the existing resources like CPU time or memory. Thus resource isolation is one way to achieve performance isolation.

Weak isolation is achieved if performance isolation is achieved within a limited number of requests sent by disruptive tenants. An overcommitted system, for example, might have the chance to restrict the load of a disruptive tenant to the amount allowed by the SLAs, but because of the overcommitment several tenants using their entire quota at one time, might infer with the other tenants.

An *unisolated* system does not provide any of the aforementioned features. This means, that tenants working within their quoata directly suffer from tenant exceeding their quota.

Examples. Google's App Engine provides automated horizontal elasticity for the application. However, it is not tenant specific and does not ensure performance isolation if elasticity is restricted. Lin's (Lin et al., 2009) approach ensures performance isolation within one single MTA.

3.4 QoS Differentiation

QoS differentiation provides one tenant another service quality than another tenant. QoS differentiation is not directly related to isolation aspects. It's easy to think of a system which can not isolate, but guarantees one tenant always better performance than another one. An example is a system where tenant A has higher thread priorities than tenant B. This ensures different response times. Nevertheless, the performance is not isolated as tenant B could still trash the system by increasing his workload. We also differ, between input and output related service differentiation, which are not mutual exclusive. The input related differentiation promises the same behavior regarding non-functional properties for all tenants by allowing different amount of workload. In the output case, the system allows the same amount of load for every tenant but differs in the output related properties, like response time.

Examples. In the academic community service differentiation in multi-tenant environments started to become of interest. Lin (Lin et al., 2009) for example provides an approach to differ response times within MTAs.

3.5 Customization

The ability to handle different tenant specific configurations regarding the UI, the systems functional/non-functional behavior and the services referenced is a key enabler for MTAs. In Kozioleks (Koziolek, 2011) architecture a separate Meta-Data Manager provides the customization information to adapt the application. Mietzner (Mietzner et al., 2009) created some patterns for multi-tenant services and service compositions. Based on these patterns they built up a service oriented system allowing extensive modifications including tenant specific developments.

We will take these two ideas to differentiate the degree a multi-tenant systems could be customized. A *configurable* application is one which provides tenant specific behavior or appearance, whereby this behavior is configured without tenant specific code. Thus, every tenant access the same code base. Such configurations might be provided by configuration files for each tenant or by tenant specific admin UIs. A change of a configuration for one tenant should not influence the behavior or appearance of the application for another tenant.

An application allowing tenant specific *code extensions* provides the most powerful way to adapt it to customers' needs. This leads to significant technical challenges in the multi-tenant scenario. Mietzner's et al. (Mietzner et al., 2008) SOA based approach of MTAs provides one way in which a tenant might replace or extend pieces of code without influencing the others.

Examples. Google Docs[4] provides office applications for private usage or companies with limited opportunities for customer specific customizing. Other SaaS providers like Salesforce provide a wide range of options including tenant specific code (Weissman and Bobrowski, 2009).

[4]https://docs.google.com/

4 MUTUAL INFLUENCES AND INTERDEPENDENCIES

This section sets the aforementioned concepts and architectural concerns in relation to each other. First, we show an overview in figure 2 followed by a textual description which contains the rationale for the interdependencies.

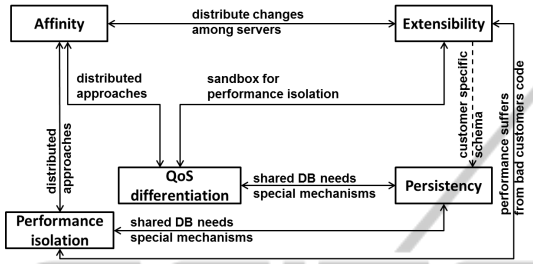

Figure 2: Interdependencies between different architectural concerns.

Basically it would be possible to use every type of affinity independently from the expected customizability, QoS differentiation or performance-isolation. However, choosing the one or the other approach might lead to more or less sophisticated implementations. The customizability feature for example requires a mechanism to deploy changes over all nodes. It is very similar to the performance-isolation and QoS differentiation aspects as both need a centralized mechanism to ensure their goals. All these aspects influence the decission to prefer the one type of affinity or the other.

The QoS differentiation's impact on the persistence layer is because one could achieve it by leveraging database features like different storage capacities. Another point is to provide every tenant its single database with different hardware settings or to use database features to prefer one tenant's requests instead of those sent by another one. The QoS differentiation implicates that one must configure the differences between the tenants, thus a tenant aware configuration is required. Beside that direct implication a particular configuration or extension might influence the tenants QoS. We do not see this as a problem for the QoS differentiation as the tenant perceives different functionality.

Extending or customizing the application significantly influences different aspects. In a non-server affine situation one has to synchronize changes in the configuration across the boundaries of the application instances. Nevertheless, solving this problem is part of the extensibility topic and not part of the affinity. Tenant specific code extensions or modifications might rely on a customer specific database schema. There are some approaches (Weissman and Bobrowski, 2009) to solve this problem without creating a separate database model. Nevertheless, one has to think about different databases for each tenant, if such extensive modifications are offered.

The decision about the chosen persistency concept might influence the level of performance-isolation which could be achieved. Independently from the tenant's restrictions at the application level, one single query on the database might cause significant performance issues for all tenants. This is why performance isolation is a major challenge when the database is shared and no isolation techniques applied on persistency. Beside that, we have the already discussed relations with respect to the customizability, the QoS differentiation and the performance impact on different affinities.

Performance isolation is influenced by many other aspects. As discussed before one has to make distributed control mechanisms available in non-server affine situations. Besides that, sandbox approaches isolating the code extensions are required as without providing sandboxes, a user could easily deploy code affecting other tenants. Another issue is the database which needs to provide adequate mechanisms to ensure isolation.

5 RELATED WORK

Multi-tenancy is a new field of research and started to get in focus with the arising of enterprise SaaS applications.

Koziolek (Koziolek, 2011) describes an architectural style of MTAs called SPOSAD based on the well-known multi-tier web application model with its architectural constraints and tradeoffs. Mietzner et al. (Mietzner et al., 2009) describes a multi-tenant architecture based on the Service Component Architecture (SCA).

We instead focus on the most important features for the tenant and its implications to some architectural concerns. Furthermore, our assumptions and restrictions regarding the architectural style are not as restrictive as presented by Koziolek.

A list of some key characteristics, a conceptual architecture of MTAs and the resulting challenges is listed in (Bezemer and Zaidman, 2010). Nevertheless, the challenges are rather discussed on a technical level and the authors did not discuss how one challenge influences conceptual architectural decisions.

There are publications (e.g. (Chong et al., 2006), (Osipov et al., 2009)) discussing how to separate the tenant's data and how to create tenant specific data

models. Wang (Wang et al., 2008) also did some performance related research. Nevertheless, there is no discussion about mutual influences of non-database and database related concepts.

Besides that, a number of publications associated with performance and resource optimization (Fehling et al., 2010), (Lin et al., 2009) exist.

6 CONCLUSIONS

In this paper we defined the concept of multi-tenancy from a vendor's point of view as sharing an application instance for several customers with their own private view onto it and motivated the approach by increased efficiency. Furthermore, we provided an overview and a discussion of concepts and architectural concerns of interest. In particular, these are performance-isolation, persistency, QoS differentiation and customizability. We presented a classification schema within each of these topics and added the discussion of different affinities. Based on the definition of these concepts we show, how these concepts how the different aspects are related.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement № 258862.

REFERENCES

Bezemer, C.-P. and Zaidman, A. (2010). Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 88–92, New York, NY, USA. ACM.

bitcurrent (2011). Bitcurrent cloud computing survey 2011. Technical report, bitcurrent.

Calvin, P. and Friedl, S. (2009). Lessons leraned: Building multitenant applications with the windows azure platform. video.

Chong, F., Carraro, G., and Wolter, R. (2006). Multi-tenant data architecture. website.

Fehling, C., Leymann, F., and Mietzner, R. (2010). A framework for optimized distribution of tenants in cloud applications. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 252 –259.

Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. (2007). A framework for native multi-tenancy application development and management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 551 –558.

Koziolek, H. (2011). The sposad architectural style for multi-tenant software applications. In *Proc. 9th Working IEEE/IFIP Conf. on Software Architecture (WICSA'11), Workshop on Architecting Cloud Computing Applications and Systems*, pages 320–327. IEEE.

Lin, H., Sun, K., Zhao, S., and Han, Y. (2009). Feedback-control-based performance regulation for multi-tenant applications. *Parallel and Distributed Systems, International Conference on*, 0:134–141.

Mietzner, R., Leymann, F., and Papazoglou, M. (2008). Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *Internet and Web Applications and Services, 2008. ICIW '08. Third International Conference on*, pages 156 –161.

Mietzner, R., Unger, T., Titze, R., and Leymann, F. (2009). Combining different multi-tenancy patterns in service-oriented applications. In *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, pages 131 –140.

Momm, C. and Krebs, R. (2011). A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings. In *Proceedings of Software Engineering 2011 (SE2011), Workshop(ESoSyM-2011)*.

Osipov, C., Goldszmidt, G., Taylor, M., and Poddar, I. (2009). Develop and deploy multi-tenant web-delivered solutions using ibm middleware: Part 2: Approaches for enabling multi-tenancy. website. http://www.ibm.com/ developerworks/ webservices/library/ws-multitenantpart2/index.html visited 23.Nov. 2011.

Schonfeld, E. (2009). The efficient cloud: All of salesforce runs on only 1,000 servers. website.

Schuller, S. (2009). What if salesforce.com werent multi-tenant? visited on 23. Nov. 2011.

Wang, Z. H., Guo, C. J., Gao, B., Sun, W., Zhang, Z., and An, W. H. (2008). A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*, pages 94 –101.

Weissman, C. D. and Bobrowski, S. (2009). The design of the force.com multitenant internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 889–896, New York, NY, USA. ACM.