Towards an Architectural Style for Multi-tenant Software Applications

Heiko Koziolek

ABB Corporate Research
Industrial Software Systems
Wallstadter Str. 59, 68526 Ladenburg, Germany
heiko.koziolek@de.abb.com

Abstract: Multi-tenant software applications serve different organizations from a single instance and help to save development, maintenance, and administration costs. The architectural concepts of these applications and their relation to emerging platform-as-a-service (PaaS) environments are still not well understood, so that it is hard for many developers to design and implement such an application. Existing attempts at a structured documentation of the underlying concepts are either technology-specific or restricted to certain details. We propose documenting the concepts as a new architectural style. This paper initially describes the architectural properties, elements, views, and constraints of this style. We illustrate how the architectural elements are implemented in current PaaS environments, such as Force.com, Windows Azure, and Google App Engine.

1 Introduction

A multi-tenant software application is a special type of hosted software that individually serves different tenants (i.e., organisations, such as companies or non-profit groups) from a single instance [CC06a]. Here, a single instance means that the software runs on the same infrastructure, is implemented from the same code base, and can be updated centrally. Currently, several platform-as-a-service (PaaS) environments, such as Force.com, Windows Azure, and Google App Engine, are emerging [AFG+09]. They are able to host multi-tenant software in large data centers and shall save software vendors costs for setting up and maintaining a hardware/software infrastructure.

Following the success of Salesforce [WB09], multi-tenant architectures have been identified as a potentially critical competitive advantage over classical single-tenant architectures in certain domains [CC06a]. However, in spite of the appearing platforms, it is still hard for software developers to design and implement efficient multi-tenant architectures [WGG+08]. Due to the missing documentation and formalisation of the underlying architectural concepts, many developers do not possess a clear view on the necessary design decisions and architectural trade-offs of a multi-tenant application.

Existing attempts on providing a structured documentation of the architectural concept

underlying a multi-tenant software architectures either depend on specific technologies (e.g., [CC06b]) or focus on restricted aspects, such as the database layer (e.g., [AGJ+08]). Descriptions of existing multi-tenant software architectures (e.g., [WB09]) help to achieve an initial comprehension but are difficult to transfer to other applications. A more abstract and technology-agnostic perspective is needed to understand the involved design decisions and architectural trade-offs.

We propose documenting the concepts and constraints underlying a multi-tenant software architecture as a new *architectural style* (the so-called SPOSAD style: **S**hared, **Po**lymorphic, **S**calable **A**pplication and **D**ata). Classical styles, such as client/server or pipe-and-filter, have been documented decades ago and are still being used to structure new software applications [TMD09]. Recent architectural styles, such as REST (REpresentational State Transfer) for the WWW [FT02] and SPIAR (Single Page Internet Application aRchitectural style) for AJAX application [MvD08] help clarifying the architectural features of web applications. In this paper, we propose documenting the features of multi-tenant software architectures as an extension of the n-tier architectural style. The described concepts shall be reusable and guide the development of new multi-tenant software applications.

The contribution of this paper is an initial description of the architectural elements and properties of a multi-tenant software architecture. We set the architectural concepts in context to the capabilities of current PaaS environments. We also provide an initial description of the design decisions to be made in the application tier and in the database tier. Ultimately, our description shall be evolved to a formal documentation of a new architectural style for multi-tenant software applications.

This paper is organized as follows: Section 2 describes three commercial PaaS environments, which are built according to multi-tenant software architectures. Section 3 then briefly recalls concepts about architectural styles that are relevant in the context of this paper. Section 4 provides an initial description of the SPOSAD style and lists architectural properties, elements, views, and constraints. Section 5 discusses the style, before Section 6 reviews related work, and Section 7 concludes the paper.

2 Multi-tenant Architectures in PaaS Environments

Several PaaS environments are currently being developed [AFG+09]. These environments are built on top of large data centers and shall help software developers to develop cloud-based applications. In the following, we will briefly summarize the most important architectural elements of three PaaS environments, before we derive an architectural style from their commonalities in later sections.

Force.com: With the force.com platform developers may build applications on top of the salesforce.com infrastructure. On a high abstraction level, the platform is built according to an n-tier architecture [WB09] comprising a presentation tier (using web browsers), an application tier, and a data tier.

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 engine generates tenant-specific application code 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. 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 [AGJ+08]). Information about tenant-specific entities is stored in an additional 'objects' meta-data table, while information about tenant-specific fields is stored in an additional 'fields' meta-data table.

Windows Azure: The Windows Azure platform by Microsoft allows deploying and running ASP.NET and WCF application in Microsoft data centers [Cha09]. The data centers run the Windows Azure Hypervisor and modified versions of Windows Server 2008 on a large number of virtual machines. The platform follows a three-tier structure.

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 application are implemented as so-called 'worker-roles'. Web-roles and worker-roles may interact asynchronously using queues. They are ideally 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 as demonstrated in [Cum09].

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, and queues for data persistency. Blobs are binary large objects up to 50 GB large, while the tables can hold a hierarchical structure of key/values pairs. Queues handle interaction between web- and worker roles by storing data portions. Tenant-specific data can either be stored in the Windows Azure storage using tenant IDs for delimination or in SQL Azure with each tenant accessing its own database instance.

Google App Engine: Google's App Engine (GAE) [Goo09] 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.

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. Developers using GAE do not gain control on the VMs running their software, which shall relieve them from the administration tasks. GAE features built-in auto-scaling, load balancing, and fail-over mechanisms between identical implementation instances in the application tier.

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.

3 Architectural Styles

To describe the architectural concepts in this paper, we use the terminology of Perry and Wolf [PW92]. 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.

According to Fielding [FT02], 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. 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 [KP88] for GUIs, REST for the WWW [FT02], and SPIAR for AJAX applications [MvD08].

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.

4 The SPOSAD Style

In this section, we first describe the essential architectural properties of multi-tenant applications (Section 4.1). Then, Section 4.2 focusses on the architectural elements of the SPOSAD style, before Section 4.3 illustrate the style with architectural views. Finally, Section 4.4 lists the architectural constraints induced by the SPOSAD style.

4.1 Architectural Properties

First, we discuss the architectural properties that relate to the goals of multi-tenant software. They mainly focus on extra-functional properties to be achieved by the style. These properties can also be viewed as requirements for a multi-tenant architecture.

Resource Sharing: the main motivation for an multi-tenant architecture is to save development and administration costs for hosted software by serving multiple tenants from the same code base and shared data repositories. The term 'resource' is used here in a broader sense. The software instance shall share hardware and software resources, such as hardware infrastructure, virtual machines, operating systems, databases, and code. On the other hand the software shall share development and maintainance resources by using a single code base for multiple tenants.

Scalability: because multiple tenants with potentially thousands of clients shall be served, scalability is an important architectural property for a multi-tenant architecture. Scalability refers to the ability of a system to either handle growing amounts of work in a graceful manner or to be readily enlarged. For example, this means that the response time of an application remains stable if the workload (i.e., the number of users concurrently using the system) is increased. 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: while many hosted, single-tenant application rely on different code bases for tenant customizations, a multi-tenant architecture relies on a shared code basis for several tenants. This feature helps to decrease the maintenance effort for the software, because bugs only need to be fixed once and updates can be installed centrally. The multi-tenant design with a shared database also reduces costs for database administration and maintenance, which does not have to be executed for each tenant.

Customizability: the ability to incorporate tenant-specific customizations is another important property of a multi-tenant architecture. Because of the shared application code base and shared database it is not trivial to allow tenants to adapt the application's business logic and data to the requirements of their clients. A well-designed multi-tenant architecture is able to find a good trade-off between resource sharing and 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.

4.2 Architectural Elements

The architectural elements described in the following are structured according to the categories by Perry and Wolf [PW92], processing elements, connecting elements, and data elements. A processing view of the style is depicted in Fig. 1 and will be explained in Section 4.3.

4.2.1 Processing Elements (Components)

Components process the data elements of the system and communicate via connectors. The following components are most relevant for the SPOSAD style.

On the client tier, users interact with the system using a client application, which can be a *web browser* for displaying HTML documents and images or a *rich client* for more sophisticated user interfaces. The client application accepts user inputs and handles the whole user interaction. It is not different from client applications in typical n-tier architectures.

On the application tier, multiple identical *application threads* execute the business logic of the application. Multiple threads or processes allow the application to scale out. It is possible to distribute the threads or processes to multiple physical processing nodes and to distribute the user requests among them. The threads can be considered polymorphic, because they may appear and behave differently for different tenants based on the meta-data they access. However, the code of the application threads comes from a single code base, which bears no tenant-specific extensions.

A *meta-data manager* handles the customization of the application threads with tenant-specific meta-data. This data may for example relate to tenant-specific input forms, UI brandings, business logic, workflows, and access privileges. There are different possibilities to implement such an application customization mechanism. For example, the meta-data manager might generate tenant-specific application code from a common code base on-the-fly. Or the meta-data manager might simply be responsible for retrieving tenant-specific meta-data and the application adapts itself to this data.

To ensure horizontal scalability, multiple application threads are running concurrently. A *load balancer* distributes client requests to these threads. Many load balancing strategies with different benefits and drawbacks are known. As the application threads shall be stateless, the load balancer can distribute subsequent requests by the same client to different application threads. The load balancer can also be responsible for auto-scaling, i.e. starting new threads upon increasing workload and stopping running threads upon decreasing workload (also known as elasticity).

On the application tier, the components processing the user requests are arranged according to a pipe-and-filter style. Therefore, additional components, such as *caches* or *proxies*, might process the user requests. However, these components can be considered optional in the SPOSAD style.

The database tier contains a *multi-tenant data resource* and a *meta-data resource*. For maximal resource sharing, a single database application should serve all tenants to save processing overheads, memory footprint, and administration costs. Different options for the data and schema layout for multi-tenant applications are discussed in the section about data elements. Hosting a large amount of tenants in the same database results in the need to partition the database and to store the data onto multiple physical nodes.

In addition to the components described so far, other optional components might be present in a multi-tenant architecture. For example, multiple tenants on the same resource require measures for user authentication and authorization and security measures (e.g., encryption). If the application shall be licensed according to a pay-per-use scheme, components

for metering application usage and billing users are required.

4.2.2 Connecting Elements (Connectors)

All components in the SPOSAD style are connected by *procedure calls*. The communication can for example follow the REST style with synchronous calls. Clients requests service from the server, the application threads carry out the service based on the data in the database and send responses back to the clients. Following the SPIAR style that is used in AJAX applications, the communication might also be asynchronous with the client requesting additional service upon state changes.

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.

The communication between the application tier and the database tier might be synchronous (e.g., for simple, short queries) or asynchronous (for long running queries). Asynchronous communication might result in more efficient resource utilization as the application threads do not have to wait for the database to respond and already serve further user requests.

4.2.3 Data Elements

The data elements in the system are the messages exchanged by the components as well as the data stored in the database tier. The *messages* sent by the components might be RESTful. Each client request at least has to include a tenant id, so that the client only sees tenant-specific data and gets a tenant-specific application.

The data stored in the database at least consists of *tenant-specific application data* and *meta-data*. When designing the data storage, the goal of the SPOSAD style is to share an adequate amount of resources.

Chong et al. [CC06b] discuss the benefits and drawbacks of different data architectures. Using a separate database per tenant is easy to implement and beneficial for security purposes, but it there is limited resource sharing and thus high costs for hardware, database administration, and backup procedures. Using a shared database, but per-tenant schemas reduces hardware costs and maintenance effort, but raises security issues and complicates backup procedures. Even more efficient is using a single schema for all tenants. Such an approach has the lowest memory overhead and low administration costs, but requires special measures for arranging the data and ensuring security. In general, the more tenants need to be served by the multi-tenant application, the better is a shared schema approach, because it reduces the memory and maintenance overheads and allows exploiting economics of scale.

Aulbach et al. [AGJ+08] have compared different schema-mapping techniques for multi-tenant databases. A *private table layout* provides a single table per tenant. In the presence of many tenants, this can induce a memory overhead for storing the individual table structures. An *extension table layout* provides a single table for common tenant data, and an additional table per tenant for schema extensions. If many tenants customize the schema,

this again results in a high number of required tables and additionally requires joins when accessing the data.

An *universal table layout* provides a single table for storing data by all tenants. This layout is for example used by the force.com architecture. It contains multiple columns without fixed datatypes for storing tenant-specific data. This layout might be more efficient, because there is no need for expensive joins to reconstruct the logical schema. However, it requires storing many null values and may require type convertions. In a *pivot table layout* each table contains tenant and row ids and a single column for a specific data type. While this layout reduces the need to store null values, it again requires joins to reconstruct the logical schema.

The SPOSAD style requires the architect to use a data architecture where resources are shared. For high scalability, the style also requires the architect to use a data partitioning scheme to physical servers that best allows for scaling out large amounts of data. This can for example involve tenant-specific partitions or local partitions for different user groups of a single tenant.

4.3 Architectural Views

Different views can illustrate the interplay of the architectural elements described in Section 4.2. Fig. 1 depicts a processing view of an architecture implemented according to the SPOSAD style. The figure shows the noticeable relation of the SPOSAD style to the n-tier architectural style, as it features a client, application, and database tier.

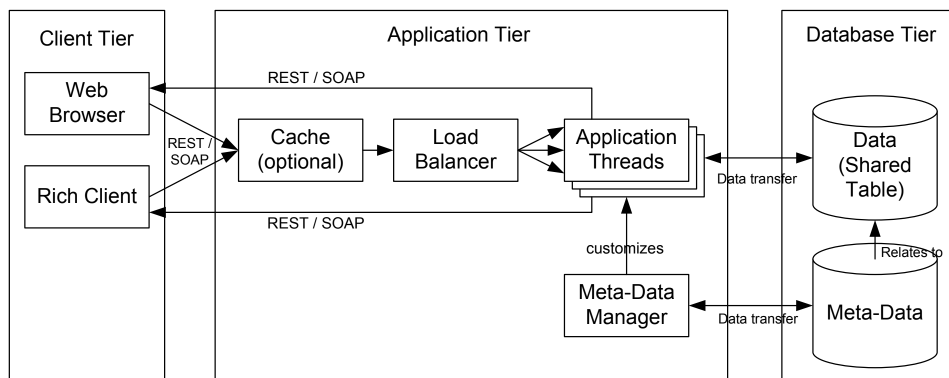

Figure 1: Processing View of the SPOSAD Style

Clients using web browsers or rich client applications interact with the application tier, which in turn accesses the database tier. The polymorphic application threads are the heart of the application tier. A load balancer directs user requests to them. The meta-data manager ensures that tenant-specific customizations are included in the application. The data tier differs from traditional n-tier architecture in the arrangement of the data, which is

stored in a multi-tenant database that allows maximal resource sharing.

The figure neglects many details of an architecture implemented according to the SPOSAD style and focuses on the elements that differentiate multi-tenant architecture from n-tier architectures. Additional components for authentication, authorization, connection pooling, security handling etc. are required. Such an application typically runs on application servers and may be hosted on virtual machines. The physical topology, i.e., the allocation of the components to hardware nodes, is also not depicted here.

4.4 Architectural Constraints

The SPOSAD style induces the following architectural constraints, which restrict architects when designing such a system.

Single code base: The application is developed in a single code base. Tenant-specific extension shall be made only via meta-data, but not by changing the code. This constraint complicates implementing the application, because mechanisms for meta-data driven tenant customizations have to be found. On the other hand this constraint enables sharing of development efforts and allows for central bug fixes and updates.

Shared resources in the database tier: Architects may not use isolated, tenant specific data layouts in the database tier as in typical n-tier applications. The style mandates sharing resources to reduce costs for database administration and backup procedures as well as hardware.

Customizable application: The application threads must allow for tenant-specific extensions 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: Client specific state, such as transactional data or inputs of users forms, may not be stored in the application tier. The application threads shall be stateless to allow serving the same client with different threads in subsequent requests. 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. Client specific state thus has to be stored at the client-side (e.g., using cookies) or in the database tier.

5 Discussion

Architects have to make several decisions and trade-offs when developing multi-tenany applications. They have to define the *degree of customization* that the application should support. More customisability 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.

Architects have to work out concepts to deal with *security issues*. 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).

When multiple tenants are using the same infrastructure, it has to be ensured that the application threads of one tenant do not interfere with application threads by other tenants (e.g., by crashing the underlying VM or decreasing performance). *Reliability measures* might include application thread replication and the isolation of performance-intensive application tasks onto individual VMs. For example, Windows Azure replicates each web and worker role (i.e., application threads) three times.

The PaaS environments described in Section 2 are already built according to the SPOSAD style or at least support building multi-tenant application and therefore should be considered by architects when making *build or buy decisions* for a multi-tenant infrastructure. Force.com includes a meta-data manager that generates the application thread code during runtime from meta-data and manages all tenant data according to a universal table layout. Azure uses web and worker roles as application threads and features a horizontally scalable storage solution with the Windows Azure tables. GAE runs application threads implemented in Java or Python code, but does not support meta-data management out-of-the-box. Like in Azure, data storage is handled in a horizontally scalable, non-relational table structure.

It is furthermore 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 software as a service solution by SAP for small companies called BusinessByDesign hosts the clients of each tenant on a dedicated physical machine. Thus, it can be considered a single-tenant solution.

6 Related Work

Due to the novelty of PaaS environments and cloud platforms, there is only limited scientific research for multi-tenant architectures.

Chong and Carraro from Microsoft discuss the business rationale of SaaS applications and describe their high level architectural concepts [CC06a]. Level 4 of their SaaS maturity model (i.e., a scalable, configurable, multi-tenant efficient application) can be considered conforming to the SPOSAD style sketched in this paper. The same authors have also discussed the benefits and drawbacks of different database layouts for multi-tenancy applications [CC06b]. However, they do not describe a reusable architectural style.

Weissman [WB09] provides an overview of the force.com architecture, which realises many concepts for multi-tenancy. His description is tied to a specific platform and thus not easily transferable to other multi-tenant architectures.

Aulbach et al. [AGJ+08] 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 completely neglect the application layer of multi-tenant applications.

Wang et al. [WGG+08] proposed a framework for implementing multi-tenant applications. They describe patterns for security, performance, and administrations isolation in such architectures and sketch customization concepts. Furthermore, they identify performance bottlenecks and optimization approaches for such applications. However, they neglect the application tier in their investigation.

Kwok et al. [KM08] 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. [MLP08] extend the service component architecture (SCA) to be able to describe multi-tenant applications.

In the area of architectural styles, Perry and Wolf [PW92] laid the foundation for describing reusable styles. Taylor et al. [TMD09] provide an up-to-date description of the most important documented architectural styles. Among them are REST [FT02] for the WWW and SPIAR [MvD08] for AJAX applications.

7 Conclusions

This paper has described the component, connectors, and data elements of a typical multi-tenant software architecture and discusses various properties and constraints of such an architecture. The description shall ultimately lead to the description of a new architectural style for multi-tenancy applications. The paper has also put the described architectural elements in context of three current PaaS environments.

The identification of a new architectural style helps developer in creating future multi-tenant software applications. While the emerging PaaS environment are well-suited for implementing such applications, there are still many design decisions at the application tier and the database tier that have to be made for each application. An architectural style can help developers in understanding the architectural trade-offs and the implications of their decisions.

As future work, we plan to formalize the style description further. We will provide different views of the style and describe further architectural constraints. Furthermore, we will analyse more existing multi-tenant applications for their implementation of the SPOSAD style concepts.

References

[AFG+09] 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. Technical Report 2009-28, UC Berkeley, 2009.

[AGJ+08] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-

tenant databases for software as a service: schema-mapping techniques. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*, pages 1195–1206, New York, NY, USA, 2008. ACM.

[CC06a] Frederick Chong and Gianpaolo Carraro. Architecture Strategies for Catching the Long Tail. Technical report, Microsoft Corporation, http://msdn.microsoft.com/en-us/library/aa479069.aspx, April 2006. Last visited 2009-10-09.

[CC06b] Frederick Chong and Gianpaolo Carraro. Multi-tenant Data Architecture. Technical report, Microsoft Cooperation, http://msdn.microsoft.com/en-us/library/aa479086.aspx, June 2006. Last visited 2009-10-09.

[Cha09] David Chappell. Introducing the Windows Azure Platform. Technical report, DavidChappell & Associates, http://go.microsoft.com/fwlink/?LinkId=158011, August 2009.

[Cum09] Cumulux. Project Riviera Website. http://code.msdn.microsoft.com/riviera, September 2009. Last visited 2009-10-09.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[Goo09] Google. App Engine. http://appengine.google.com, October 2009. Last visited 2009-10-09.

[KM08] Thomas Kwok and Ajay Mohindra. Resource Calculations with Constraints, and Placement of Tenants and Instances for Multi-tenant SaaS Applications. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 633–648, Berlin, Heidelberg, 2008. Springer-Verlag.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[MLP08] Ralph Mietzner, Frank Leymann, and Mike P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 156–161, Washington, DC, USA, 2008. IEEE Computer Society.

[MvD08] Ali Mesbah and Arie van Deursen. A component- and push-based architectural style for AJAX applications. *J. Syst. Softw.*, 81(12):2194–2209, 2008.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[WB09] Craig D. Weissman and Steve Bobrowski. The Design of the Force.com Multitenant Internet Application Development Platform. In *Proc. 35th SIGMOD International Conference on Management of Data (SIGMOD '09)*, pages 889–896, New York, NY, USA, 2009. ACM.

[WGG+08] Zhi Hu Wang, Chang Jie Guo, Bo Gao, Wei Sun, Zhen Zhang, and Wen Hao 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)*, pages 94–101. IEEE, 2008.