Software Architecture Patterns for System Administration Support

ROLAND BIJVANK, HU University of Applied Sciences, Utrecht, the Netherlands
roland.bijvank@hu.nl

WIEBE WIERSEMA, HU University of Applied Sciences, Utrecht, the Netherlands
wiebe.wiersema@hu.nl

CHRISTIAN KÖPPE, HAN University of Applied Sciences, Arnhem, the Netherlands
christian.koppe@han.nl

Many quality aspects of software systems are addressed in the existing literature on software architecture patterns. But the aspect of system administration seems to be a bit overlooked, even though it is as important as other aspects. In this work we start with mining software architecture patterns that, when applied by software architects, support the work of system administrators. We present five patterns: PROVIDE AN ADMINISTRATION API, SINGLE FILE LOCATION, USE BUILT-IN SYSTEM LOGGING, CENTRALIZED IDENTITY MANAGEMENT, and MULTI-TENANT APPLICATION.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design—*Design Patterns*

General Terms: Software Architecture, System Administration

Additional Key Words and Phrases: Software Architecture, System Administration

ACM Reference Format:

Bijvank, R., Wiersema, W., Köppe, C. 2013. Software Architecture Patterns for System Administration Support – L(October 2013), 14 pages.

1. INTRODUCTION

In the article *A plea from sysadmins to software vendors: 10 Do's and Don'ts* by Thomas Limoncelli [Limoncelli 2011], system administrators collected a basic list of do's and dont's for software vendors in order to make the life of the system administrators more easy.

For a large number of points in this list there is a high agreement between administrators on what the best practices should be. However as system adminstrators are on the "receiving" end for a new or modified application, it is necessary to influence other parties who have a key position in the creation or the changing of an application.

A role that fits this key position is the software architect. Among other concerns the software architect is responsible for the software architecture. The software architecture is the main design document for the software of an application. The design decisions taken in that document have a profound impact on the workload of the system administrators.

This paper aims at influencing the software architects and the software architecture by providing patterns for software architecture that are endorsed by system administrators.

The focus of Software Architecture is often on realizing quality attributes, such as those described in ISO 25010: Functional suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability and Portability. Many patterns have been described, e.g. in the POSA book series [Buschmann et al. 1996], and their general applicability

for realizing the qualities has been discussed [Harrison 2011]. There are also publications that focus on patterns for specific quality aspects, like patterns for fault tolerant systems [Hanmer 2007] or security patterns [Schumacher et al. 2005]. But there is one quality attribute where not much attention has been paid to: Portability and its sub-qualities Adaptability, Installability, and Replaceability. A number of concerns from system administrators are covered by the aforementioned attributes, but the mapping of the concerns on the attributes is not intuitive.

There have been several initiatives to describe patterns from the perspective of a system administrator, but these are mainly focused on infrastructure and middleware. Examples of these initiatives are:

—Daniel Jumelet: Open Infrastructure Architecture repository (OIAr)[1] - this site provides a wide variety of infrastructure patterns for several working areas: Client Realm, Middleware, Network, Security + Support, Server, Storage. Beside this repository also contains architecture & design guidelines in the form of construction models at various levels and from various angles. It is constructed by making use of one of the most important tools of OIAm: The Building Blocks Model. The Building Blocks Model is primarily a *decomposition* tool. That means that it is used to dissect infrastructure landscapes into logical dimensions and parts in order to enable structured and methodological modeling (*composition*).

—Gregor Hohpe and Bobby Woolf: Enterprise Integration Patterns[2] - this site provides a consistent vocabulary and visual notation to describe large-scale integration solutions across many implementation technologies. It also explores in detail the advantages and limitations of asynchronous messaging architectures.

Both approaches — software architecture patterns for realizing the above described quality attributes and patterns that support the work of system administrators — don't touch some important aspects of the intersection of software architecture and system administration. Therefore we want to introduce a set of patterns which bridges this gap, based on the needs of the system administrators.

The problems that are cited in the aforementioned article have been experienced within daily system administration practice.

With these patterns we want to give ideas to software architects and application developers on how to improve their applications from a system administration viewpoint.

2. THE PATTERNS

In this paper we present the first four software architecture patterns for system administration support:

—PROVIDE AN ADMINISTRATION API

—SINGLE FILE LOCATION

—USE BUILT-IN SYSTEM LOGGING

—CENTRALIZED IDENTITY MANAGEMENT

—MULTI-TENANT APPLICATION

The patterns use a version of the Alexandrian pattern format, as described in [Alexander et al. 1977]. The first part of each pattern is a short description of the context, followed by three diamonds. In the second part, the problem (in bold) and the forces are described, followed by another three diamonds. The third part offers the solution (again in bold), consequences of the pattern application — which are part of the resulting context — and a discussion of possible implementations. In the final part of each pattern, shown in *italics*, we discuss related patterns and offer a rationale for the pattern based on literature.

[1] http://www.infra-repository.org/oiar/index.php/Main_Page
[2] http://www.eaipatterns.com/

PROVIDE AN ADMINISTRATION API

The system to be built includes the possibilities of being configurable, whereby configuration files alone are not sufficient. These configurations are often administered by special employees, like application administrators, and not the core-users of the system itself.

❖ ❖ ❖

If the administrative interface is a GUI, many of the standard administration tasks can not be automated. Repetitive tasks have to be completed again and again, which leads to a high frustration of the administrators. It also can be hard to get remote access to such a GUI.

Unexpected usage. System administrators have their own ways of organizing their administration tasks. The strive to automate many parts, often in unexpected ways, and a GUI is minimizing the possibilities of doing so.

Admin OS vs. System OS. The operating systems which admins are using for their administration tasks often differ from the OS the application to be administered is running on. Providing an GUI as administrative interface often means that this GUI is only executable on certain OS's, which certainly restricts system administrators in an unnecessary way.

❖ ❖ ❖

Therefore: Provide an API for all required administration functionality. Make this API externally available, easily accessible and well documented, so that admins can automate administrative tasks and integrate it easily in the administration processes.

Offering an API for the administrator provides much more flexibility to the system administrators for administering the systems in the way they think fits best. It gives them enough freedom to integrate the administration in existing processes. In order to be able to offer this high degree of freedom regarding the usage of the API, the system developers have to carefully design it and to offer the administration functionality in appropriate abstraction levels. This means that the API should be fine-grained enough.

Tools for automation can make use of the administration functionality if they can connect to the provided API. For example, the right API helps to automate tasks that are part of a new employee account creation process. [Limoncelli 2011].

There might be some security issues when exposing administrative features. Making use of a PROXY [Buschmann et al. 1996] can help here. The PROXY can include an authentication mechanism and block all unauthorized access attempts, this will be discussed in more detail in the implementation description below.

If the system evolves, then also the API is likely to change which might require adaptations the system developers are not aware of. This is a general problem in interface- and component-based development and needs to be addressed in the design of the API too.

Providing an API might require a good documentation, whereas an administrative GUI can be more intuitive and self-explaining. For example: an API might require the correct spelling of user roles which need to be assigned to new users. A GUI can offer a selection list including all user roles and possibly an extra explanation of these roles in an apart window section. This minimizes the need for extra documentation. The API should therefore include an extensive help, containing all information necessary for using the provided administration functionality. For the same reason the API should include a good exception handling in combination with clear error messages.

In the most simple cases the pattern is a specific variant of a SERVICE LAYER [Fowler 2002]. In this case it does not contain any logic, but simply forwards all requests to already existing subsystems that offer the administration functionality. This is shown in Figure 1.

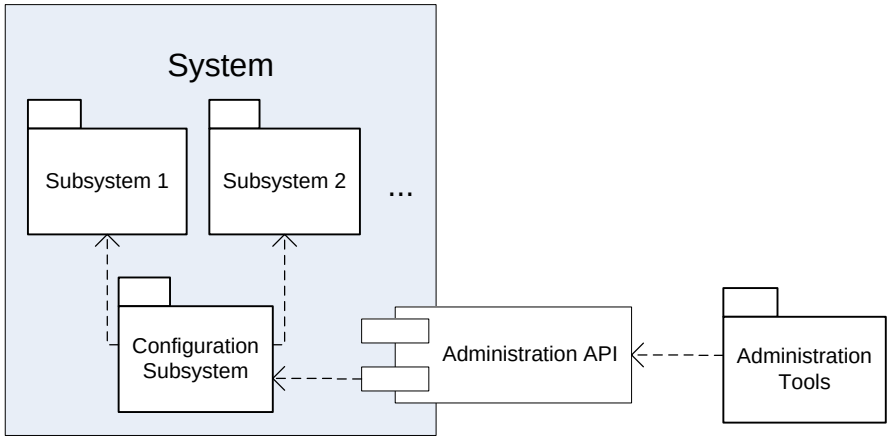

Fig. 1. Main solution structure of PROVIDE AN ADMINISTRATION API

If the administration API should not be publicly available due to security reasons, a PROXY [Buschmann et al. 1996] could be used to adequately address this issue. Figure 2 shows the main design. The protection proxy needs to include some mechanism for authentication and authorization of the requester. These can be implemented making use of e.g. pattern ADAPTER [Gamma et al. 1994] because this pattern can influence the visibility of the administration API with respect to authentication and authorization of the requester.

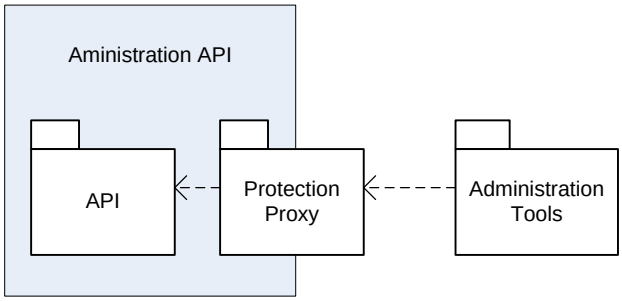

Fig. 2. An administration API including a protection proxy for security reasons

In certain cases the implementation language of the system and that of the administration API are different. Main reason for this could be that the administration API is required to be provided in a specific scripting language that suits the administrators' tasks best. In that case the administration API subsystem also becomes a specific kind of an ADAPTER [Gamma et al. 1994] between these two implementation languages.

The problem of different platforms used for the system and in the administration environment can be minimized by making use of cross-platform scripting languages like Python, Ruby or TCL. This is also a certain advantage above graphical administration interfaces, as it removes the platform-specific issues caused by the GUI technologies. In

combination with such a cross-platform scripting language this pattern shows its real strength as one can uniformly approach the administration API on any given platform.

Ideally, any changes in the system itself do not lead to changes in the administration API. However, if also functionality of the system regarding its configuration is changing, then also the API likely needs to be changed. The tools of the administrators are dependent on the API both syntactically and semantically in varying degrees. Unfortunately are both dependency types interrelated: the less syntactic the dependency is, the higher it is semantically and vice versa. One criterion that can be used for determining if the API should decrease the syntactic or the semantic dependencies is how easy it is to adapt the connection to the API on either syntactic and semantic level. If the interfaces are easy to adapt on both sides, then one should prefer more syntactically dependent interfaces that explicitly contain the semantic information in the naming of the methods and parameters. If the interfaces are not easy to adapt, then the syntactical dependencies should be low by using more generic interfaces that merely require different parameter contents but no interface adaptations.

One possibility of implementing this administrative API in the Java programming language are Java Management Extensions[3] *(JMX).*

[3] `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`

SINGLE FILE LOCATION

Files are an old and established mechanism used by applications to store and retrieve configuration, libraries, state, data etc. Most applications use files in their own unique manner and store files in various locations.

❖ ❖ ❖

Having such an extensive use of files can cause system administrators to have difficulty in finding the files necessary for their tasks during the life cycle of an application. Especially so if the files are dispersed over different folders or hidden in system-folders of the Operating System. System administrators also want to be able to perform version control on the files, having files files spread over the whole disk practically negates to ability to perform versioning and baselining of the configuration of an application.

Special cases that cause much aggravation:

—Distributed Applications.
Many applications consist of different subsystems, which often require subsystem-specific administration tasks. These subsystems are in many cases developed by different teams, resulting in dispersed groups of similar artifacts for each subsystem. This situation is well suited for developers as they can work in parallel. During deploy or system administration activities

—Hard-coded Locations.
This is the case when the developers put the location of the configuration files in source code and provide no parameters or interface to influence this location. This means the path can only be changed by building and deploying a new version of the application. Running multiple instances of a program on a machine with different parameters is effectively blocked by this approach. Additionally it can pose security risks if the file location is in a privileged location such as `C:\Program Files` for Windows based systems.

—Non Human Readable Configuration Files.
When an application provides a Graphical User Interface for configuring the application, it happens that such an application stores the captured configuration in a non human readable file. The disadvantage of this approach is that the system administrator can only see the configuration by starting the application and opening the dialogue to see the settings. This also blocks automation of deploy and install scripts and integration with automatic deploy tooling such as Puppet[4] or Chef[5].

❖ ❖ ❖

Therefore: Put all related files in one (hierarchical) location. Make the path of this location configurable.

Files that logically belong together and should be at the same location are: the binaries of a system, the configuration files and the data files. In the case of log files one should first consider to USE BUILT-IN SYSTEM LOGGING.

❖ ❖ ❖

Rationale.
Without using this pattern the files of applications will be dispersed over several distinct locations which makes it hard to maintain the application. When a file of a module isn't used anymore it will easily remain in disuse and get overlooked which causes pollution of your hard disk.

[4] `https://puppetlabs.com/`
[5] `http://www.opscode.com/chef/`

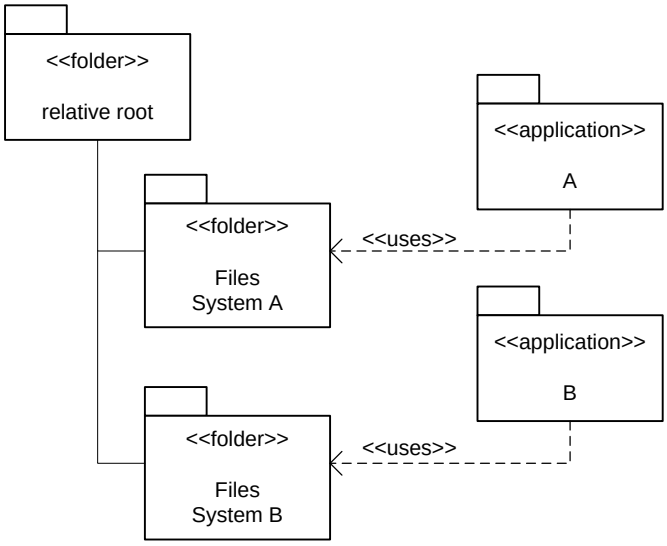

Fig. 3. Basic example

Fig. 4. Another example

USE BUILT-IN SYSTEM LOGGING

The application needs to provide the ability of logging certain events or actions by using the built in system logging of a platform.

❖ ❖ ❖

Having a variety of logging formats and log-file locations makes it hard to monitor the state of a whole enterprise, including all running applications.

Format Variety. A high variety of logging formats increases the complexity of integrating the information held within those several log files. It becomes a burden to nullify the different lay-outs of these log files.

Location Variety. When having a variety of log file locations the dispersion of those locations makes it difficult to gather those files to one stack.

❖ ❖ ❖

Therefore: Use the built-in system logging mechanism or define a standard format to be used by all systems.

Don't reinvent the wheel. Many monitoring tools use the system built-in logging mechanisms so don't try to circumvent it. This allows the administrators to make use of existing tools that collect, centralize, and search the logs [Limoncelli 2011].

If it is not possible to use the built-in system logging, e.g. because of different operating systems being used, then define a standard for your system landscape and ensure that this is used for logging. Combine this approach with SINGLE FILE LOCATION.

❖ ❖ ❖

Besides the above mentioned reasons, it is a lot easier to automatically generate incidents from specific defined events from the built-in system log for an IT service management (ITSM) tool. This ITSM tool can be configured to forward the automatically generated incidents directly, without human intervention, to the second line specialists.

CENTRALIZED IDENTITY MANAGEMENT

also known as: IDENTITY MANAGEMENT BUS.

The system makes use of user identities which need to be managed.

❖ ❖ ❖

Decentralized user identity management means a lot of extra work as identities have to be managed on many different places and it is hard to get a centralized overview of all existing or available identities. This also makes role management much more complex.

Separation of Duties. Especially when Separation of Duties (SoD) is a concern such as within financial environments it is important for organizations to be able to show to e.g. an EDP auditor that all regulations are fulfilled.

❖ ❖ ❖

Therefore: Make use of a centralized identity management system if this is available.

This solution has several advantages: if the centralized identity management system (CIM) is also connected to the human resources system (HRM)-system, it is easier to revoke certain grants due to retirements etc. Also user roles could be (automatically) inferred from function profiles in the HRM system.

When no centralized identity management system is available a lot of organizations make use of something like Active Directory Services (ADS). Mostly in these cases where ADS is used this isn't connected to a HR system whereby the events or triggers for the HR processes placing in, leave service, function change or department change are missed in the ADS. This causes an increase in maintenance activities to take care of pollution of the ADS.

❖ ❖ ❖

There is an urgent need within medium to large organizations to centralize role based access information. Several applications have role based access information. This dispersion of information leads to a high maintenance sensitivity. Which demands a high level of deployment. Therefore the dispersion of information needs to be centralized within a solution according to this pattern.

MULTI-TENANT APPLICATION

A multi-tenant application is a shared solution (i.e. HRM) used by different tenants ((client) organizations, departments). It is a single application with scalable resources to meet the performance demands of tenants.

❖ ❖ ❖

Many companies are looking for a scalable architecture to deal with burst loads or for an approach for sharing. Virtualizing your hardware seemed one solution but isn't realy scalable in both directions (upwards and downwards)

❖ ❖ ❖

Therefore: Make use of multi-tenant applications to reduce hardware investments or to outsource one's software and hardware to a SaaS/PaaS-provider.

This solution has several advantages: Combining virtualization, elasticity and multi-tenancy results in optimized usage of data center resources as it means CPU, memory and network resources are maximally deployed.

When no multi-tenancy is used a lot of organizations make use of virtualization and/or elasticity.

❖ ❖ ❖

Besides above mentioned advantages multitenant applications are typically required to provide a high degree of customization to support each target organization's needs. Customization typically includes the following aspects:

—Branding: allowing each organization to customize the look-and-feel of the application to match their corporate branding (often referred to as a distinct "skin").

—Workflow: accommodating differences in workflow to be used by a wide range of potential customers.

—Extensions to the data model: supporting an extensible data model to give customers the ability to customize the data elements managed by the application to meet their specific needs.

—Access control: letting each client organization independently customize access rights and restrictions for each user.[6]

In addition to the above mentioned extensions for the data model also some tenant specific extensions to the application itself are sometimes needed. Several patterns can be used for this e.g.: COMPOSITE, DECORATOR [Gamma et al. 1994]. Beside these patterns also SOA based approach of Multi-tenant applications can realize such a MTA-architecture. A new architectural style is needed (the so-called SPOSAD style: Shared, Polymorphic, Scalable Application and Data):

Figure 5 shows the inherent relation of the SPOSAD style to the n-tier architectural style, as it features a client, application, and database tier. 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.

Requirements for a multi-tenant architecture are [Heiko Koziolek, 2010?]:

—Resource Sharing: hardware and software resources, such as hardware infrastructure, virtual machines, operating systems, databases, and code.

[6] http://en.wikipedia.org/wiki/Multitenancy

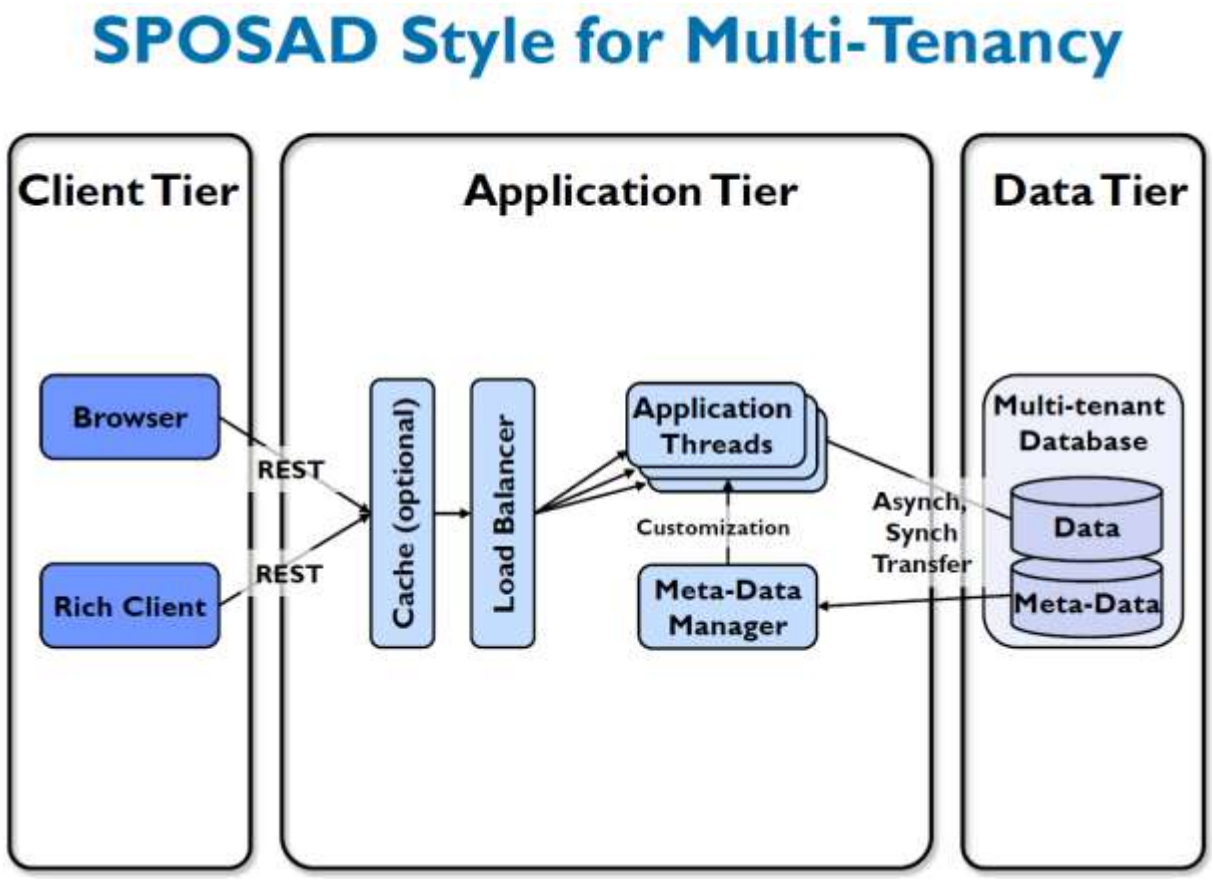

Fig. 5. SPOSAD Style for Multi-Tenancy

—Scalability: 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: 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 multitenant design with a shared database also reduces costs for database administration and maintenance, which does not have to be executed for each tenant.

—Customizability: ability to incorporate tenant-specific customizations. 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: the user interface shall be configurable through tenant-specific customizations. It allows different tenants to create their own branding for an application.

One of the implementation issues that comes from the aforementioned requirements is to have a database table column which holds the tenant ID.

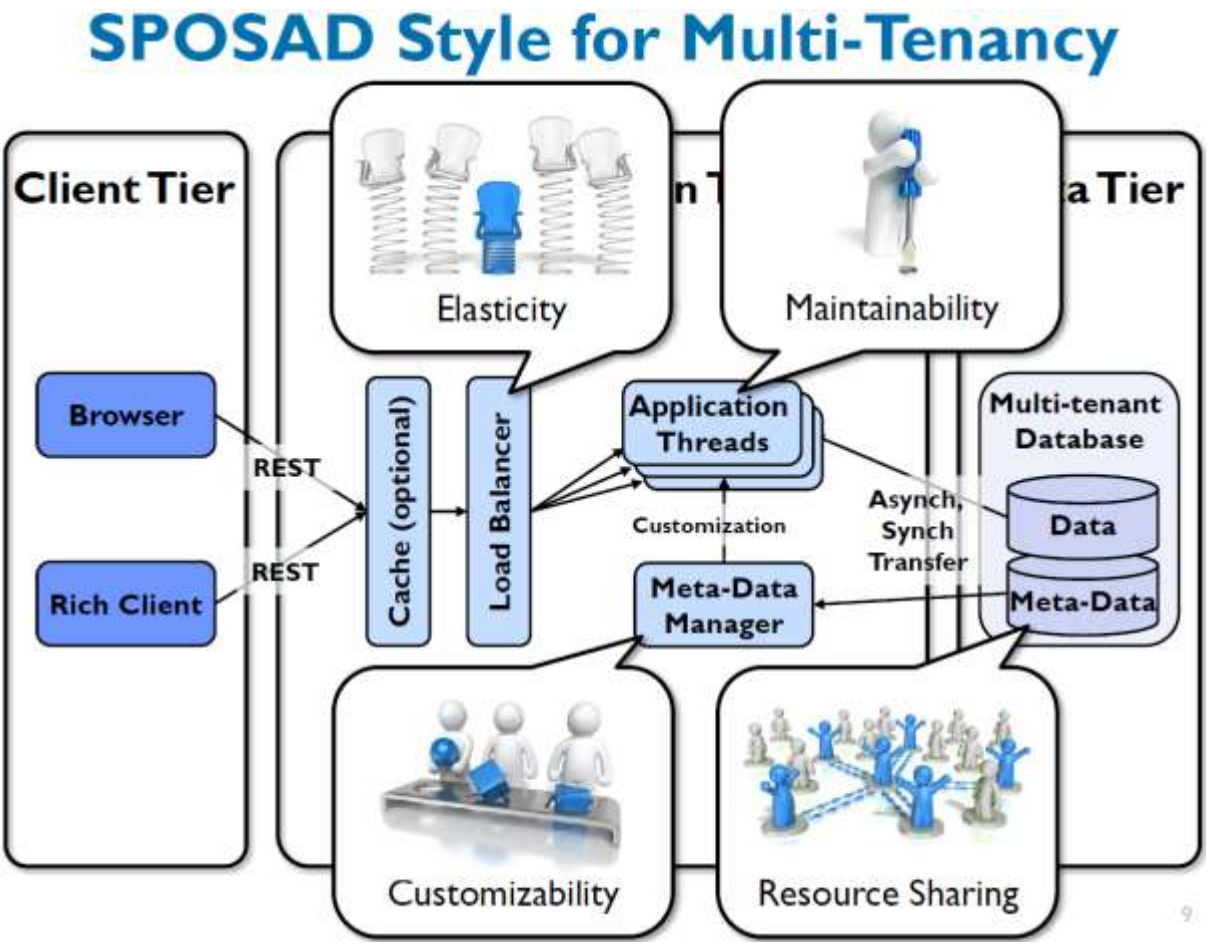

Fig. 6. SPOSAD Style for Multi-Tenancy - Architectural properties

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 metadata 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.

3. ACKNOWLEDGEMENTS

todo

REFERENCES

ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. 1977. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester.

FOWLER, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA.

HANMER, R. 2007. Patterns for Fault Tolerant Software.

HARRISON, N. B. 2011. Phd. Ph.D. thesis, Rijksuniversiteit Groningen.

LIMONCELLI, T. A. 2011. A plea from sysadmins to software vendors: 10 Do's and Don'ts. *Communications of the ACM 54,* 2, 50–51.

SCHUMACHER, M., FERNANDEZ, E., HYBERTSON, D., AND BUSCHMANN, F. 2005. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons.