Institute of Architecture of Application Systems

Combining Different Multi-Tenancy Patterns in Service-Oriented Applications

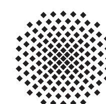
Ralph Mietzner, Tobias Unger, Robert Titze, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
http://www.iaas.uni-stuttgart.de

in: Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009).
See also BibTeX entry below.

BibTeX:

```
@inproceedings {INPROC-2009-50,
    author = {Ralph Mietzner and Tobias Unger and Robert Titze and Frank Leymann},
    title = {{Combining Different Multi-Tenancy Patterns in Service-Oriented Applications}},
    booktitle = {Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009)},
    editor = {IEEE Computer Society},
    publisher = {IEEE},
    pages = {131--140},
    month = {Oktober},
    year = {2009},
    isbn = {978-0-7695-3785-6},
    doi = {10.1109/EDOC.2009.13},
    keywords = {multi-tenancy; SaaS; services; SOA; composite applications},
    language = {Englisch},
    cr-category = {D.2.11 Software Engineering Software Architectures,
                   H.4.1 Office Automation},
    ee = {http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5277698},
    department = {Universit{\"a}t Stuttgart, Institut f{\"u}r Architektur von Anwendungssystemen},
    url = {http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-50&engl=0}
 }
```

© 2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.



Universität Stuttgart
Germany

Combining Different Multi-Tenancy Patterns in Service-Oriented Applications

Ralph Mietzner, Tobias Unger, Robert Titze, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de

***Abstract*—Software as a service (SaaS) providers exploit economies of scale by offering the same instance of an application to multiple customers typically in a single-instance multi-tenant architecture model. Therefore the applications must be scalable, multi-tenant aware and configurable. In this paper we show how the services in a service-oriented SaaS application can be deployed using different multi-tenancy patterns. We describe how the chosen patterns influence the customizability, multi-tenant awareness and scalability of the application. Using the patterns we describe how individual services in a multi-tenant aware application can be not multi-tenant aware while maintaining the overall multi-tenant awareness of the application. We show based on a real-world example how the patterns can be used in practice and show how existing applications already use these patterns.**

***Keywords*-multi-tenancy; SaaS; services; SOA; composite applications**

I. Introduction

One benefit of service oriented architectures (SOA) is the facilitation of the construction of new applications as they can be recursively assembled out of existing services. Considering the principles of SOA, in addition, reuse of existing services is simplified. Reusing existing services prevents the tedious and error prone repetition of the construction of functionality that has been done multiple times before. Web service technology as one technology stack to implement a SOA allows defining reusable components as so-called "Web services". These services can be implemented in any programming language that supports the Web service stack. For this purpose Web services provide a uniform description and access scheme in heterogeneous infrastructures, which allows accessing a Web Service independently from its concrete location.

Location transparency and reuse foster new delivery models for software ranging from infrastructure as a service (IaaS) over platform as a service (PaaS) to software as a service (SaaS). Companies move away from solely running their applications in their own data centers (*on premise*) but more and more applications are outsourced to third party providers. Based on Web service technology services are consumed or offered to other companies, either free of charge or for a pre-defined amount of money. Using service oriented architectures and for example Web service technology individual services that make up an application can be run in different delivery models. For example one service in an application could be a service run in a SaaS model as a third party, while another service is run on an infrastructure provided in an IaaS delivery model by another provider. A third service of the can be run on-premise.

Today, Web services are provided by several Web applications such as E-Bay or Salesforce and partially are offered via search engines such as Seekda[1]. Other services come with standard products such as SAP or are developed from scratch in an enterprise as part of a home-grown system. While building new *composite* applications (applications composed out of a set of services), application vendors can decide either to develop their own services or to reuse existing services provided by different providers.

Considering the different types of services mentioned above, we can distinguish them in two main aspects: The first aspect is the aspect of outsourcing: Is the service under my control or is it run by someone else? The second aspect is the multi-tenancy aspect. Is the service shared between multiple customers (tenants) or is it run solely to be used by me? How do these aspects influence the creation of new applications, how can services run at third party providers be integrated into new applications? In which delivery model can a new application be offered given the services it orchestrates? In the remainder of the paper we will investigate these questions. We begin by an investigation of existing service delivery models and related work in Section II. By means of a case study from the automotive sector we motivate the need for modeling support for the different types of services and their composition in Section III. Having done so we introduce and describe three basic multi-tenancy patterns for services in Section IV and introduce a taxonomy of multi-tenancy patterns. In Section V we investigate how services of different delivery patterns can be combined. We describe how concepts and patterns known from enterprise application integration (EAI) can be adapted and used to integrate services with different multi-tenancy patterns into new applications. We show how we have applied the patterns in other projects in Section VII and finish with an outlook and a description of future work.

[1] http://www.seekda.com

II. Background and Related Work

Currently, many IT-vendors propose SOA as basic architectural style for building application infrastructures and many reference architectures based on the vendors' specific SOA products are published [22]. As a consequence of the adoption of Web service technology more and more services such as the GlobalWeather Service by WebservicesX[2] or the ZipCode Service by StrikeIron[3] become available in public. These services can be reused in new applications without any prior configuration or subscription. For example the GlobalWeather service exposes a WSDL that describes its interface and no other information is needed to include it in new applications. Other hosted services such as E-Bay and Salesforce expose Web services that allow third party applications to integrate with their Web applications. Salesforce, for instance, allows customers to connect to the Salesforce application via two different WSDL interfaces. The first WSDL interface (the so-called "enterprise WSDL") is specific for one customer and contains all customizations that the customer has made. The second WSDL interface (the so-called "partner WSDL") is a generic interface that is metadata driven and abstracts from the concrete customizations of one particular customer. It is therefore suited to be used by multi-tenant aware applications that themselves have different tenants mapping to different tenants in the Salesforce application.

Regarding the development and deployment of multi-tenant aware applications, several related work exists. In [7] a framework for multi-tenant aware SaaS applications including data isolation, performance isolation or configuration is described. In [18] an infrastructure is presented that allows the integration of SaaS applications with other applications. Other work on application architectures (such as the one introduced in [4] describe how multi-tenancy can be achieved across different layers of the application, particularly regarding the database layer.

Many of these infrastructures have in common that the runtime infrastructure is centralized and shared by many applications. Moreover, these infrastructure are providing infrastructure services which can be reused by any application (e.g. security services) [7], [14]. However, they require the whole application to be multi-tenant aware while our approach allows parts of the application to be multi-tenant aware and parts not, depending on the services the application is composed of. Our approach can integrate services that are deployed on one of these platforms. Other platforms exist that allow the composition and automated deployment of Web service based applications such as [2], [11], [17]. Our work differs from these approaches as we explicitly take multi-tenancy into account and focus more on the modeling than on the deployment, at least in this paper.

In [4] four SaaS maturity levels are introduced, relating to how the SaaS application is delivered to many customers. In level 1 an application is specifically run for one customer at an SaaS provider. This level corresponds to the traditional ASP [19] model. Starting from level 2 the SaaS application is customizable via configuration but still in level 2 one instance of the application serves only one customer. At level 3 a single instance of the SaaS application serves multiple tenants. Separate configuration data is used to configure the application to the need of each separate customer. In order to achieve level 4, the SaaS application is developed as a single instance multi-tenant application and several instances are run in a load-balanced server farm. At level 4 load is balanced via a tenant-load-balancer that spreads the load over the various servers the application runs on. We show how these levels can be realized for individual services in a service-oriented application. We furthermore describe patterns on how to solve several multi-tenancy problems e.g. how to make a service aware about which tenant invoked it.

Pattern languages have been first introduced in architecture [1] to describe recurring problems and nuggets of advice on how to solve these problems. Several books on patterns exist in computer science [6],[5],[9] that describe advice on how to solve recurring problems in several domains of software engineering. In this paper we use the notion of a pattern language to describe the multi-tenancy patterns for services that we first outlined in [13]. We also introduce mechanisms on how to integrate services deployed in one of the multi-tenancy patterns. These mechanisms are based on enterprise application patterns introduced in [9]. Inspired by [9] each multi-tenancy pattern below is given (i) a title, (ii) an icon, (iii) a context that describes the problem that can be solved using the advice described in the pattern, (iv) a short question that summarizes the problem the pattern solves, (v) a description of the forces that make the problem a problem, (vi) the result when the solution is applied, and optionally (vii) relations to other patterns, (viii) examples on how to implement the pattern and examples for services using this pattern.

III. Running Example

In this section we describe a real-world composite SOA application that serves as an example for the patterns and algorithms introduced in this paper.

The eCommerce Concept (eCCo) [20] application is an automotive point-of-sales application developed together with the automotive industry. The application deals with the sale of new and used vehicles at a dealer. It is able to create an offer for a customer, and calculate the prices according to the vehicle configuration. In case of new vehicles the car can then directly be ordered, whereas used vehicles can be picked from a pool of available vehicles.

The application is hosted in a central data center and can be used by various dealers of a manufacturer. As big car manufacturers typically have dealers in various markets

[2]http://www.webservicesx.net
[3]http://www.strikeiron.com/

across the world, the application needs to take the different requirements of these markets into account. Therefore the individual distributors in different markets can customize the application to the needs of the specific market. For example they might offer additional financing options, set tax rates or modify the application to add additional signature steps. To cater for that variability, the application is built using a service oriented architecture where services are realized as Web services that run on an IBM WebSphere application server. These Web services are orchestrated using BPEL. BPEL processes are run on an IBM WebSphere Process Server. To adapt the application for different markets, the dealer organization of that market can modify configuration values (such as the tax rate in that market). We therefore regard each dealer organization for a market as a tenant of the eCCo application. In case more elaborate modifications are needed, services can be exchanged for market specific services, or external services (for example banking services) can be integrated into the application. Customizability of the application also affect the process and GUI layer. For example the Spanish market requires an additional signature step over the German market, that must be configured.

IV. Service Tenancy Patterns

In this section we introduce three basic patterns related on how services are shared between different tenants. A tenant for a service can be a sole user or a whole company consisting of a set of users. In case of our running example each dealer organization for a different market is a tenant of the application. In case of Salesforce, each company that has an account (with possibly hundreds of users) is regarded as a tenant. All following patterns apply to services that are part of an application or that are so-called *external services*. External services, or outsourced services are not part of the application package but and are not under direct control of the application. These external services are provided by a third party provider, which can reside inside or outside the own company.

A. Taxonomy

In order to build multi-tenant aware applications out of services, multi-tenancy support is required in the whole life-cycle of a service. In this paper we focus on two steps of the life-cycle: development, and deployment. During development of a service, built-in support for multi-tenancy can be archived by introducing configuration options for tenants. While deployment, services instances can be created and distributed over the infrastructure according to the multi-tenancy requirements of the application.

As a prerequisite for defining multi-tenancy patterns we ordered the different life-cycle aspects using a taxonomy. Figure 1 shows the taxonomy of services regarding their multi-tenancy patterns. Regarding the observations made in the introduction and the related work, services can either be configurable or non-configurable as shown in Figure 1.

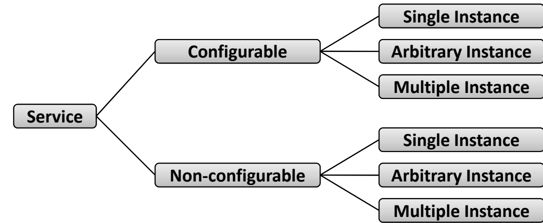


Figure 1: Taxonomy of Patterns

Configurable services offer the possibility to customize a service according to the tenant's needs. E.g. the user interface service of the eCCo application can be configured with the national language of the dealer or the invoice service may be configured to consider local tax rates.

Services can be further classified according to the relation between a service instance and a tenant. The straightforward possibility may be to reuse a single service instance for each tenant. Accordingly, only for configurable services a new configuration has to be deployed. In this case only a single instance of each service of an application would run that is used by each tenant. However, a vanilla single instance approach can not solve all requirements. In the eCCo example e.g. the car configuration services can be reused by all tenants. In contrast, the service managing the customer data is obliged to be separated for each dealer in order to be compliant with local data privacy rules. This can be achieved by deploying a dedicated instance of the customer data service per tenant. Furthermore, even for the car configuration service it can make sense to deploy several instances. In the case that the existing instance of the service is overloaded it may be easier to deploy a new service instance instead of adapting the existing instance in order to handle the additional load.

As a consequence, services can be classified in three instance types: *single instance*, *arbitrary instance*, and *multiple instance*. In case a service is non-configurable and the same instance of the service can be used by multiple tenants we call the service a *single instance service*. In case a service is non-configurable and for each tenant a separate instance of the service is deployed we call the service a *multiple instances* service. In case a service is configurable and one single instance of the service serves all tenants we call this service a *single configurable instance* service. There is also the possibility to deploy a service following the *multiple configurable instances* service, in case a service is configurable but nevertheless a new instance is needed for each tenant.

The *arbitrary instances* service pattern is a mix of the single and multiple instances service and is therefore not regarded explicitly below. Arbitrary instances means that some tenants share a single instance of the service, but others do not. This might be due to quality of service requirements or

regulatory requirements that prevents a subset of the tenants to share a service with other tenants.

B. Service Patterns Assumptions and Properties

Service tenancy patterns describe how multi-tenancy and per-tenant configuration and customization can be achieved using services in SOA-based applications. The following patterns apply to both atomic and composite services. In case we talk about an *instance* of a service we mean an instance of the application implementing the service. This means that the code for a service is only deployed once and can be used for several tenants. We assume that one instance of a service can also be a *virtual instance*, i.e. that several instances exist, but are virtualized by an ESB, or load balancer [3]).

In order to support visual modeling using multi-tenancy patterns we created an icon for each pattern defined below. Figure 2 shows all pattern icons.

In the following pattern descriptions we describe the properties of the patterns with regard to (i) the degree of customizability, (ii) the isolation of data, (iii) the ease of deployment and update , (iv) the difficulty of development (i.e. do developers need to take special care for multi-tenancy) and (v) the scalability with regarding to tenants (i.e., the ability to distribute the load of several tenants on one service instance). Table I shows a summary of the properties. We assign a (+) in the respective category if the pattern is best suited to fulfill the goal in the category. An (+/-) is given if the goal is partially solved and a (-) is given if the goal of the category is not solved. The abbreviations used for the patterns are the following: single instance (single), single configurable instance (single conf.) and multiple instances (multiple). How the individual marks are given is explained in the pattern descriptions below.

	single	single conf.	multiple
customizability	-	+/-	+
isolation	-	+/-	+
update	+	+/-	+/-
development	+	-	+
scalability	+	+/-	-

Table I: Summary of service tenancy patterns

C. Single Instance Service

Context: A service is the same for all tenants that trigger its invocation. If the application is updated, the service should be updated only once for all tenants.

How can I implement a service that has the same behavior for all tenants?

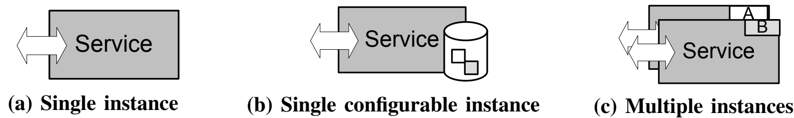
Forces: A service should have the same behavior for all tenants. It is not necessary that the service "knows" which tenant triggered its invocation, thus the service can run in a standard non tenant aware environment. The service should be updatable at once for all tenants which prevents deploying the same service under different endpoints for each tenant. The service and underlying middleware should be deployed only once for all tenants to allow the sharing of load from different tenants to maximize the resource utilization of the underlying middleware and hardware.

Solution: Use a *single instance service* (cf. Figure 2a) that is deployed once for all tenants. The service can be deployed and updated for all tenants at once, thus making installation and update of the service easy.

Result: A service following the single instance service pattern is deployed once for all tenants. Therefore the service shows the same behavior for all tenants and is not customizable on a per-tenant basis. As the single instance service does not distinguish between different tenants all data used in the service is shared between all tenants. Modification of the service for all tenants is made easy by using this pattern As long as only the service implementation is modified, i.e., to resolve errors or to change an underlying database, the new service can be modified and rolled out once and can be used by all tenants. In case a modification of the service interface is necessary, all tenants that call the service must be updated. The single instance service does not require any special consideration regarding multi-tenancy, therefore existing services that have not originally been developed for multi-tenant aware applications can be easily reused and developers do not need to take any special precautions. The single instance service scales well as it can be deployed several times and the load of several tenants can be balanced among these service deployments.

Next: The single instance service has the same behavior for all tenants. In case tenant specific behavior is needed the single configurable service pattern (cf. section IV-D) can be used. It allows a single service to behave differently for different tenants based on configuration data. Additionally, the multiple (configurable) services pattern (cf. section IV-E) is another alternative. In this pattern tenant specific behavior is realized by deploying a different service for each tenant. The single instance service pattern can be called under a *tenant context*, i.e., a tenant invoking the service is identified by a unique identifier (e.g. included in the SOAP header). The service middleware can then exploit the tenant context to do non-functional processing such as authorization or billing. However, invocation under tenant context is not needed by this pattern, as a single instance service cannot distinguish different tenants.

Examples: A Web service that realizes a basic non tenant-specific functionality of an application, such as the transformation of units from metric to imperial, can be realized using the single instance service pattern. The single service pattern is predestined to be an external service that is reused from other applications or third party vendors. An example for such a service is the GlobalWeather service from WebservicesX.



(a) Single instance (b) Single configurable instance (c) Multiple instances

Figure 2: Service icons

In our running example the "basic car details" service is realized as a single instance service. This service returns the details (such as basic specifications) for a given type of car from a central database.

D. Single Configurable Instance Service

Context: A service should have tenant-specific behavior. It should be easy to add new tenants to the service. Additionally updating of the non-tenant specific parts should be as easy as updating a service following the single service pattern (cf. section IV-C). Furthermore the service should scale well with the number of tenants, because load of different tenants can be balanced among the instances of the service.

How can a service with tenant specific behavior be deployed only once for all tenants?

Forces: In case a service is deployed following the single instance service pattern (cf. section IV-C) it has the same behavior for all tenants that use it. A service should behave differently for different tenants. New tenants must be added without recompiling and redeploying the service.

Solution: Use configuration data to provide tenant specific behavior for single instance services. In [4], [7] the term of configuration data is used to describe data that is used by an SaaS application to provide tenant specific behavior. Tenants can easily be added by adding new tenant-specific configuration data so that no redeployment of the service is needed when a new tenant is deployed.

Result: A *single configurable instance* (cf. Figure 2b) service uses configuration data to provide tenant-specific behavior. Configuration data can either be deployed locally for the service (i.e. in configuration files in the deployment directory for the service) or can be deployed in a central configuration database [4], [7]. In order to know which tenant's configuration must be used, the single configurable instance service must be invoked using the invocation under tenant context. The service implementation must then associate the tenant context with the right configuration data. Additionally the service implementation must ensure that a tenant cannot access data of another tenant, i.e. via data-isolation techniques in the database [4]. Since the service must be invoked under a tenant context it is possible to monitor which tenant has used the service and perform tenant-specific billing.

As all tenants use the same instance of the service the non-tenant specific parts can be updated at once for all tenants which yields to the same results as with the single instance pattern. However, updating parts that allow tenant-specific configuration is harder as it requires the redeployment of the configuration data for each tenant. The development of single configurable instance services needs to take multi-tenancy into account from the beginning as both isolation and configuration requirements demand that the service explicitly knows about the concept of a tenant. In terms of scalability the single configurable instance service behaves similar to the single instance service. However, in case several instances are spread over different machines and are served by a load-balancer, the configuration data for each tenant must also be made available to all these instances.

Next: Tenant specific behavior for a service can also be achieved by employing the multiple instances service pattern (cf. section IV-E). However, an advantage of the single configurable instance service pattern is that the service can be updated in one central location. Using the multiple instances service pattern updating a service is harder since it involves updating all services separately. Furthermore the single configurable instance service pattern is multi-tenant aware as a service using this pattern can be used by different tenants. This is especially important since the load for the service can be balanced between multiple tenants. In case a service is deployed using the multiple instances pattern the underlying infrastructure must always be provisioned for peak load of the one tenant using a certain instance of the service.

Orchestration and process engines (such as WS-BPEL engines) often do not support the configuration of process instances on a per-tenant basis (i.e, a differently configured process instance is carried out based on which tenant started the process instance). In addition to ensuring isolation of the runtime on a per-tenant basis a multi-tenant aware workflow engine must ensure that audit data for a tenant cannot be accessed by another tenant. Until such engines become commonly available, services realized as business processes (for example specified in WS-BPEL) should be deployed using the multiple instances pattern (Section IV-E), so that a different process model is deployed for each tenant.

Examples: A Web service following the single configurable instance service pattern could for example be implemented in Java using JAX-WS [8] as follows: JAX-WS (and other Web service stacks) allow using the concept of handlers to process a SOAP [21] message before it is handled over to the respective service. Handlers can for example read SOAP headers (such as a SOAP header containing a tenant context)

and perform processing based on these handlers. A *tenant handler* can then retrieve tenant specific metadata for example from a database or a properties file and put it in the message context. The message context is then passed to the service implementation. The service implementation can then read the tenant specific configuration data (such as tenant-specific database tables or tenant-specific business rules) from the message context. New tenants are deployed by deploying new configuration data and associating it with the tenant id. However, the customizability of services developed with programming languages such as Java or C# is limited to the possibilities given by the language (such as properties files or dynamic class loading in the extreme case). In case more complex customizations are needed the *multiple instances service pattern* can be used. A real life example for such a service is the Web service exposed by Salesforce. This Web service behaves differently depending on the configuration data for each individual tenant.

In our running example the "calculate vehicle price" service is a single configurable instance service. Each market can specify it's own tax numbers as well as specific rules on how to compute the vehicle price (including different financing options). These are stored in a configuration database. The service is implemented only once and accesses this configuration database to retrieve the appropriate values. Additionally the GUI of the application is configurable via configuration metadata to include the different vehicle financing options of different markets.

E. Multiple Instances Service

Context: An application requires tenant-specific behavior for a service. The service implementation is very specific for each tenant or the underlying middleware does not support multi-tenancy.

How can I realize tenant-specific behavior when the service logic is very specific for each tenant?

Forces: Sometimes it is not feasible to use the single configurable instance service pattern to achieve tenant specific behavior. Reasons for that might be that the service logic is completely different for different tenants and cannot be derived only by configuration. Another reason is that the service handles privacy critical data and the service must be kept in a secure zone that is only accessible by one tenant. Additionally, the underlying middleware (such as a BPEL engine) is not able to handle multiple tenants for one instance of the service. Other quality of services that are specific for one tenant may mandate the use of the multiple instances pattern too. For instance one tenant needs reliable messaging while others do not but need better response-times, therefore several instances of one service can exist (maybe on different middleware) with different qualities of service.

Solution: Use the *multiple instances service* (cf. Figure 2c) pattern to achieve tenant-specific behavior. Each tenant, or groups of tenants use their own service that realizes the specific behavior they need.

Result: Several services are deployed that perform the same basic task in the SaaS application but with tenant specific behavior. The services itself can be realized as normal non tenant aware services in case one service is deployed per tenant. As each tenant is served by its own instance, data-isolation is not an issue as long as proper authorization is in place (i.e. each instance can only be accessed by the correct tenant). However, the multiple services pattern corrupts the notion of a single instance multi-tenancy architecture. This is the case, because a service instance does not serve all tenants but only a subset of them or only one in the extreme case.

While updating non-tenant-specific parts of a service deployed using the single configurable instance service is easy since only one service instance needs to be updated, updating services following the multiple services pattern is harder. Updating such services requires to perform updates on each service implementation separately. In order to partially overcome this limitation *templates* can be used. Templates define basic parts of a service and variability points [10],[13],[12] similar to those in product line engineering that can be modified separately for each tenant. Updating all services is then quite similar to the single configurable instance service pattern. It involves updating the basic template and then using the template and the tenant specific configuration data to re-generate the multiple services. We introduce the *multiple configurable instances* pattern to denote that a multiple instances service is derived by configuration from a template.

The development of multiple instance services is easier than that of single configurable instance services as each service instance only serves one tenant. Therefore multi-tenancy does not need to be taken into account explicitly during development and non-multi tenant-aware services can be reused by simply installing them separately for each tenant.

A major drawback of the multiple instances service is that the load on a service cannot be balanced between several tenants, instead, each instance of a service deployed using the multiple instances pattern must be provisioned for peak load of the respective tenant. Another drawback of the multiple instances pattern is that the deployment of new tenants becomes harder as a new service must be deployed for each tenant instead of only configuration data. In case the application is spread over several environments, the service must be deployed on every environment to ensure that load can be balanced over all environments using a load balancer. The third problem relates to the evolution of the SaaS application.

Next: In cases where services must be completely different for each tenant or cannot be shared for regulatory or technical reasons, the multiple instances service pattern can be used instead of the single configurable instances pattern (cf. section

IV-D) to achieve tenant-specific behavior.

In our running example the "sign offer" Web service is realized as a multiple instances service. Unlike in other countries, customers from Spain have to sign an offer for a vehicle as a notice of intent. To realize this variable part, a special service (along with the corresponding GUI) is deployed in multiple instances mode specifically for the "Spain" tenant. All other tenants use a shared service that does not contain the extra signature step.

V. Combination of Service Tenancy Patterns

In Section IV-B we described the basic multi-tenancy patterns in which services can be deployed. When building composite applications out of these services the patterns are important to determine how the services need to be invoked. In the left part of Figure 3 a BPEL process P that is deployed in multiple-instances mode needs to invoke a Web service S1 that is deployed in single configurable instance mode. The service S1 needs to be invoked under a so-called *tenant context* (indicated by the box on the invocation arrow) which contains information about which tenant invoked it.

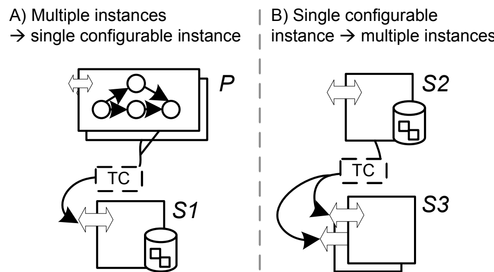


Figure 3: Combination of different patterns

Therefore each instance of P must send the tenant context (TC) when invoking the service S1. On the right part of the figure the scenario is reversed. A service deployed in single configurable instance mode (service S2) invokes a service deployed in multiple instances mode (service S3). S2 includes a tenant context in the invocation message. A component is now needed that reads the tenant context and distributes the message to the right instance of service S3.

A. Service Invocation

Integrating different services with different interfaces is a classical enterprise application integration (EAI) problem [9], [15]. In the EAI community, EAI patterns [9] are a well-known toolbox that offers hints on how to solve integration problems. In this paper we do not describe how to integrate services that have different interfaces or data formats, but how to integrate services that are deployed following different multi-tenancy patterns. As motivated in Figure 3 different mechanisms are needed to do so. We introduce these mechanisms in detail below. The mechanisms below can again be seen as patterns, but due to the limited space in this paper we refrain from presenting them in a formal pattern format as the service tenancy patterns above.

B. Invocation under Tenant Context

Tenant context is a mechanism to submit information about a tenant in the invocation message of a service.

The tenant context can simply be a tenant identifier or complex authentication information. The tenant context can be made explicit (for example in a SOAP header) or can be contained implicitly in the payload of the message (for example through a vendor id). However, it is important that the invoking and the receiving service have agreed on a format for the tenant context that both recognize. We call a message to a service that contains a tenant context, an *invocation under tenant context* (cf. Figure 4).

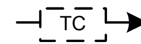


Figure 4: Invocation under Tenant Context

C. Basic Invocation

We call a normal invocation of a service (e.g. via a SOAP message) *basic invocation* to distinguish this invocation from the invocation under tenant context (cf. Figure 5).

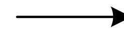


Figure 5: Basic Invocation

D. Tenant Context-Based Router

A tenant context-based router (cf. Figure 6) is derived from the *content-based router* introduced in [9].

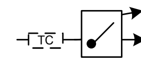


Figure 6: Tenant context based router icon

The tenant context-based router routes an invocation message that contains a tenant context to a concrete endpoint depending on the tenant whose context is in the message. E.g. in Figure 3 B a tenant context-based router could be used to route the invocation message to the right instance of service S3.

E. Tenant Context Appender

A tenant context appender appends a tenant context to an invocation message. Situations where this is necessary is for example the situation depicted in figure 3 A) where a multiple instances component sends out a basic invocation message that needs to be augmented with a tenant context. The tenant context is important so that the receiving service can distinguish which tenant has sent the invocation message. In some cases the tenant context can be already included in

the application In some cases, however, the tenant context needs to be appended outside of the application, e.g. when the application cannot be modified. In these case the tenant context appender can be used.

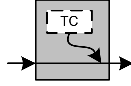


Figure 7: Tenant context appender icon

F. Tenant Context Wrapper

A special case of the tenant context appender is the *tenant context wrapper* (cf. Figure 8). The tenant context wrapper completely encapsulates the tenant context handling from a service. For example a tenant context wrapper can be used in conjunction with a BPEL process that is deployed using the single instance service pattern. The tenant context wrapper extracts the tenant context from the invocation of a BPEL process instance and adds it to all invocations that are triggered by that instance. This allows the BPEL process to call services deployed using the single configurable instance pattern without changing the process model. Figure 8 shows such a tenant context wrapper for a single service.

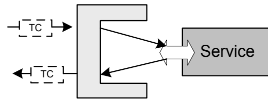


Figure 8: Tenant context wrapper icon

VI. Supporting the Modeling of Composite Applications Using Multi-Tenancy Patterns

In this section we describe the different combinations of patterns that are possible and how they can be integrated using the patterns and mechanisms described above. We first begin by listing all possible combinations and then derive an algorithm that automatically inserts the correct mechanisms to support a modeler in the modeling of composite applications. As a result a modeler can then purely focus on wiring the functional services, while additional components such as tenant context-based routers or tenant context appenders are inserted automatically upon deployment of the application.

A. Possible Combinations

Table II shows how the two basic invocation patterns *basic service invocation* (b) and *service invocation under tenant context* (tc) can be used to permit arbitrary combinations of service tenancy patterns in an application. Furthermore the table shows how tenant context appenders (ap) and tenant context routers (r) can be used to combine different service tenancy patterns. For example line 3 column 4 reads (tc+r) which means that a client deployed using the single configurable instance pattern can invoke a service deployed using the multiple instances pattern by performing an invocation under tenant context (tc) that is then distributed to the service by a tenant context based router (r). A star (*) annotated to an invocation denotes that in case the invoked service recursively invokes a service that requires invocation under tenant context, the basic invocation might not be enough and an invocation under tenant context in conjunction with a *tenant context wrapper* must be used.

In order to determine whether basic invocation can be used or if invocation under tenant context is needed a so-called *service invocation graph* can be computed. The service invocation graph contains all services as nodes and all invocations as directed edges. Two types of edges exist, edges of type *b* (requiring basic invocation) and edges of type *tc* requiring invocation under tenant context. Again an edge pointing from one node to another denotes that the source node issues the first invocation of the target node (service). However the conversation between the two can be of any arbitrary complex message exchange pattern. The service invocation graph can then be traversed bottom-up to determine whether a service invocation needs to be under tenant context because the target service is deployed using the multiple instances or single configurable instances pattern or any combination involving these two patterns. Note that the outgoing edge of a *tenant context appender* as well as all incoming edges of a *tenant context based router* and *single configurable instance* services must be of type *tc*.

client/service	**single**	**single conf.**	**multiple**
single	b/b+ap*	b+ap	b+ap+r
single conf.	b/tc*	tc	tc+r
multiple	b/tc*/ b+ap*	tc/b+ap	b/tc*/ b+ap*

Table II: Combination of different patterns

B. Modeling Support

An application developer, however does not want to model the specific invocations as well as routers and appenders needed to invoke a service under tenant context. As these are not relevant for the application from a functional viewpoint, the application developer should be freed from the burden of needing to specify them explicitly.

Given the different combinations of patterns in Table II corresponding tooling can automatically insert the necessary tenant context appenders and routers. These are then used to insert a tenant context into an invocation message or to route messages depending on their tenant context.

The application developer can therefore simply wire components with different patterns. Upon deployment the deployment infrastructure then rewrites the application in a way that it inserts the necessary routers and appenders. Therefore the deployment infrastructure traverses the *invocation graph* bottom up. As described in Section VI-A this traversal allows determining which invocation must be under

tenant context and which not, (cf. Table II). The algorithm can then insert new appenders and routers where needed (and where the developer has not originally added them).

C. Customizable Process

Our automotive point-of-sales application introduced in Section III basically consists of a set of BPEL business processes that orchestrate a set of Web services. The left part of Figure 9 shows the general schema of the application. Each tenant is served by the same instance of the business process that retrieves configuration data (such as tax rates or financing options) from a configuration database for each tenant. Figure 9 shows two scenarios. In the left scenario

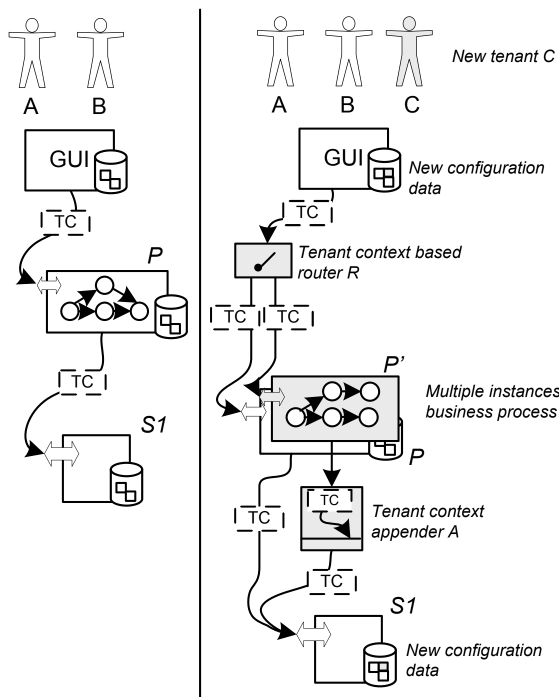


Figure 9: Customization of Processes

the business process customization is done via the single configurable instance pattern. However, tenant C in the second part of Figure 9 needs advanced customization of the business process and therefore opts to deploy the process model in the multiple instances pattern. In our use-case this is for example necessary for the Spanish distribution organization that needs additional confirmation steps that other countries do not need. This has impact on the whole application. For example between the GUI and the business process a *tenant context based router* R needs to be inserted that routes the message to the right endpoint. This is either the process P using the single configurable instances pattern, or the process P' using the multiple instances pattern. Service S1 is deployed using the single configurable instances pattern and therefore must be invoked under a tenant context. This tenant context must be inserted into the invocation message of the multiple instance process P' by the *tenant context appender* A.

VII. Validation and Implementation

As described in Section III we applied the patterns in a real-world case study in the automotive sector. There we showed how the different patterns can be applied to an enterprise application that is offered as a SaaS application by one branch of the company to other departments of the same company. In the EaaS project (EAI as a Service) [16] users can model and combine executable EAI patterns into an integration solution in a graphical tool. These patterns are then annotated with multi-tenancy patterns (depending on user requirements). Upon deployment the patterns are then transformed into executable BPEL and Web service code [15], [16]. The algorithm as described in VI-A is then applied and corresponding routers and appenders are inserted (depending on the multi-tenancy patterns of the individual patterns). Afterwards a deployment process is run that deploys the BPEL processes and necessary services so that the integration solution can be run. Additionally we implemented a sample application based on Web services that are orchestrated by a BPEL process to further validate our approach. Users can choose between predefined BPEL processes and can set some configuration properties for them that are retrieved from a configuration Web service. In addition users can upload their own process definition (with the same interface as the predefined processes) that is then deployed using the multiple instances pattern. We are currently extending this sample application into an on-demand process platform named DecidR, where users can create and use their own business processes that orchestrate a set of pre-defined services such as a human task service or e-mail services. Documentation and first prototypes can be found at the project Website[4].

VIII. Conclusions and Outlook

In this paper we introduced and evaluated a set of patterns that can be used to design, develop and deploy process-aware service-oriented SaaS applications. We describe how multi-tenancy of the whole application can be achieved, while individual services exploit maximum customizability. We describe patterns how individual services of an SaaS application can be deployed with different degrees of customizability and how these different patterns can be combined. SaaS customers benefit from our approach as they can completely customize certain parts of an SaaS application while reusing other parts of the SaaS application. SaaS providers benefit from the approach as they can extend their user base to customers that can not be satisfied with the customizability of pure multi-tenant applications by relieving the multi-tenancy paradigm

[4] http://code.google.com/p/decidr/

for parts of the application. In future work we will describe a multi-tenant aware infrastructure for services and processes and how tenant deployment scripts can be generated using the patterns.

Acknowledgment

This work is partially funded by the ALLOW project . ALLOW (`http://www.allow-project.eu/`) is part of the EU 7th Framework Programme (contract no. FP7-213339).

References

[1] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] R. Anzböck, S. Dustdar, and H. Gall. Software configuration, distribution, and deployment of web-services. In *Proc. SEKE 2002*.

[3] D. Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., 1st edition, June 2004.

[4] F. Chong and G. Carraro. *Building Distributed Applications Architecture Strategies for Catching the Long Tail*. MSDN architecture center, http://msdn2.microsoft.com/en-us/library/aa479069.aspx, 2006.

[5] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Nov. 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. *Addison-Wesley Professional Computing Series*, 1995.

[7] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. *CEC/EEE 2007*, 2007.

[8] M. D. Hansen. *SOA Using Java(TM) Web Services*. Prentice Hall PTR, May 2007.

[9] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, Oct. 2003.

[10] M. Jaring and J. Bosch. Architecting product diversification - formalizing variability dependencies in software product family engineering. In *QSIC '04*, 2004.

[11] Z. Maamar, Q. Z. Sheng, and B. Benatallah. On composite web services provisioning in an environment of fixed and mobile computing resources. *J. Information Tech. and Mgmt, Special Issue on Workflow and e-Business, Kluwer*, 5, 2004.

[12] R. Mietzner and F. Leymann. Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. In *SCC 2008*, 2008.

[13] R. Mietzner, F. Leymann, and M. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-Tenancy Patterns. In *ICIW*, 2008.

[14] J. Sathyan and K. Shenoy. Realizing unified service experience with SaaS on SOA. In *COMSWARE 2008.*, 2008.

[15] T. Scheibler and F. Leymann. A Framework for Executable Enterprise Application Integration Patterns. In *I-ESA*, 2008.

[16] T. Scheibler, R. Mietzner, and F. Leymann. EAI as a Service - Combining the Power of Executable EAI Patterns and SaaS. In *EDOC 2008*, 2008.

[17] Q. Z. Sheng, B. Benatallah, Z. Maamar, M. Dumas, and A. H. H. Ngu. Enabling personalized composition and adaptive provisioning of web services. In *In Proc. CAiSE 2004*.

[18] W. Sun, K. Zhang, S. Chen, X. Zhang, and H. Liang. Software as a Service: An Integration Perspective. *Service-Oriented Computing ICSOC 2007*, 2007.

[19] L. Tao. Shifting paradigms with the application service provider model. *Computer*, 34(10):32–39, 2001.

[20] R. Titze. Web Service Deployment Strategien (in German). Diploma thesis, University of Stuttgart, 2009. http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2852.

[21] Weerawarana, S. et al. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[22] M. Zirn. Application upgrades and service oriented architecture. Technical report, Oracle, 2008.

All hyperlinks in this document last checked on March 5th 2009