2009

Service Objects: Adaptable, Metadata-Based Services for Multi-Tenant On-Demand Enterprise Applications

Sebastian Enderlein
Hasso Plattner Institute, sebastian.enderlein@hpi.uni-potsdam.de

Marco Helmich
Hasso Plattner Institute, marco.helmich@hpi.uni-potsdam.de

Juergen Mueller
Hasso Plattner Institute for Software Systems Engineering, juergen.mueller@hpi.uni-potsdam.de

Jens Krueger
Hasso Plattner Institute, jens.krueger@hpi.uni-potsdam.de

Vadym Borovskiy
Hasso Plattner Institute, vadym.borovskiy@hpi.uni-potsdam.de

See next page for additional authors

Authors
Sebastian Enderlein, Marco Helmich, Juergen Mueller, Jens Krueger, Vadym Borovskiy, Alexander Zeier, and Hasso Plattner

Service Objects: Adaptable, Metadata-Based Services for Multi-Tenant On-Demand Enterprise Applications

Sebastian Enderlein
Hasso Plattner Institute
sebastian.enderlein@hpi.uni-potsdam.de

Marco Helmich
Hasso Plattner Institute
marco.helmich@hpi.uni-potsdam.de

Jürgen Müller
Hasso Plattner Institute
juergen.mueller@hpi.uni-potsdam.de

Jens Krüger
Hasso Plattner Institute
jens.krueger@hpi.uni-potsdam.de

Vadym Borovskiy
Hasso Plattner Institute
vadym.borovskiy@hpi.uni-potsdam.de

Alexander Zeier
Hasso Plattner Institute
alexander.zeier@hpi.uni-potsdam.de

Hasso Plattner
Hasso Plattner Institute
hasso.plattner@hpi.uni-potsdam.de

ABSTRACT

An adaptive, standardized service layer is a key feature of a multi-tenant on-demand enterprise application. Custom business logic and data need to be exposed via services that are tailored to the respective customer organization. Ideally, this layer of web services can be automatically derived from the underlying domain model. This paper aims to describe means to design and implement such a service layer by following a lean, model-driven approach based on the runtime interpretation of metadata. Finally, the implementation will be validated against a real-world show case.

Keywords

Software as a Service, On-Demand, Service-Oriented Architecture, Model-Driven Architecture, Web Services, REST, Multi-Tenancy, Enterprise Software, Enterprise Resource Planning

INTRODUCTION

Software-as-a-Service (SaaS) is on the way up. Gartner forecasts the annual growth rate for SaaS enterprise applications at 22.1% through 2011, more than double the growth rate of the overall enterprise software market (Mertz, Eschinger, Eid, Pring, 2007). The shift to on-demand software promises low recurring subscription fees instead of barely calculable upfront costs and additional charges for maintenance, hardware, and professional services for adaptation. The high cost-saving potential makes SaaS enterprise software interesting for Small and Medium-size Enterprises (SMEs) that cannot afford the financial investments that come with professional on-premises enterprise software. Nonetheless, even SMEs appear to have complex, unique business processes that are the key to their success (Fink and Markovich, 2008). Consequently, a central requirement to an enterprise application is a high degree of flexibility.

By choosing to offer an enterprise application as a service that is consumed on demand, SaaS vendors will undergo major changes in their own business model. A centralized hosting and operation of the application offers some cost-saving potential through economies of scale. However, the pricing pressure will ultimately force the SaaS vendor to further lower the operating costs by implementing multi-tenancy (Chong and Carraro, 2006), the ability to manage multiple customer organizations in a single application instance. The result is a hosted, large-scale enterprise application that incorporates a high degree of resource sharing between customer organizations (tenants).

As Chong and Carraro (2006) describe further, the two major requirements, multi-tenancy capabilities and flexibility, are diverging. In our recent paper "Customizing Enterprise Software as a Service Applications: Backend Extension in a Multi-tenancy Environment" (Mueller, Krueger, Enderlein, Helmich, Zeier, 2009), we described a way to enable flexibility in terms of custom, tenant-specific business logic in a multi-tenant backend. In this paper in turn, we will now focus on means to

expose this tenant-specific business logic and data through a custom-tailored service layer to make additional features remotely consumable in the first place.

RELATED WORK

Software as a Service has been first introduced by Bennett, Layzell, Budgen, Brereton, Macaulay, and Munro (1999) who described an upcoming shift towards service-orientation and virtual market places. Sääksjärvi, Lassila, and Nordström (2007) explain and evaluate the SaaS business model and identify possible risks, challenges, and benefits for stakeholders.

Chong and Carraro (2006) provide a high-level description of SaaS application architectures, discuss several degrees of multi-tenancy and flexibility-related issues, and identify new markets to be addressed through SaaS. Guo, Sun, Huang, Wang, and Gao (2007) introduce a framework for multi-tenant applications and present approaches to isolate tenants from each other through a Multi-Tenancy Enablement Layer.

Schulz-Hofen (2007) introduces *Webdata*, a middleware to expose an object-oriented domain model as RESTful web resources. Hirschfeld and Kawamura introduce an approach to implement dynamic and adaptable web services based on Aspect-oriented Programming. Wirdemann and Baustert (2007) describe means to implement REST-based web services with RubyOnRails.

A SERVICE-ORIENTED ARCHITECTURE TO EXPOSE THE TENANT-SPECIFIC DOMAIN MODEL

Every company has its own, unique set of business processes and data. Fowler defines the total of these elements as the *Domain Model* of the respective enterprise (Fowler and Rice, 2006). The elements of this domain model are represented by Business Objects (BOs) that model real-world entities related to the business as software artifacts. As described in (Mueller et al, 2009), we see BOs as (object-oriented) objects that contain data and define operations on this data. We implemented a domain model in Ruby and used *Mixins* to allow third party developers to implement tenant-specific modifications of BOs without impacting the common code base or business logic and data of other tenants. While the execution of custom business logic and the ability to store additional data are major milestones towards a flexible, multi-tenant enterprise application, a major aspect has been left out, yet. How do we make (additional) business logic and data accessible to the customer?

The state-of-the-art answer to this question is the implementation of a *Service Layer* (see Figure 1), which "defines an application's boundary with a layer of services that establishes a set of available operation and coordinates the applications' response in each operation" (Fowler and Rice, 2006).

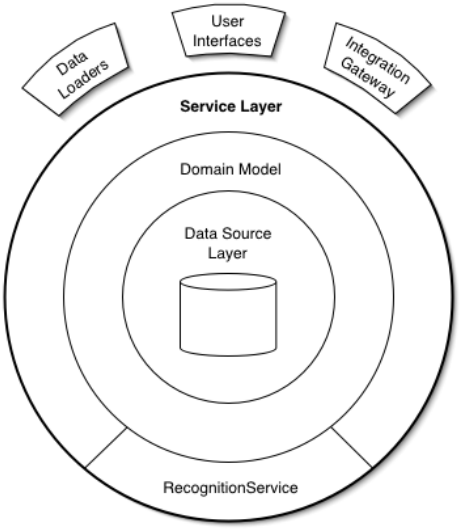

Figure 1. A Service Layer to expose the Domain Model (Fowler and Rice, 2006)

While the implementation of a service layer is important in an on-premises enterprise application, it is essential to the success of a SaaS solution. The whole product becomes worthless to customers if they are not able to remotely access their data and processes in a standardized, comfortable manner.

Problem Statement

The multi-tenant architecture of the SaaS enterprise application has major impact on the design and implementation of the service layer. While the application instance and major elements of the infrastructure are shared among tenants, each customer organization requires custom-tailored services that expose its unique domain model and data. Consequently, the application has to ensure that only the specific customer is able to consume the services, and that modifications of the service layer at no circumstances affect other tenants. Furthermore, if the domain model of a specific tenant is modified, the surrounding service layer needs to be adapted. A tenant-specific coding for each service leads to high redundancy and additional implementation effort for partners in the context of a large-scale SaaS enterprise application with thousands of customers. Thus, a model-driven approach based on a declarative notation to describe the elements of the domain model on service level is required. The application in turn must be capable of interpreting the given descriptions at run-time, whenever requests from the respective tenant are received.

Show Case

In order to illustrate the concepts that will be proposed, our considerations are framed by a real-world problem of a company with 300 employees. The company sells folding facades, grew during the last years, now feels the need to implement an Enterprise Resource Planning (ERP) system, and is attracted by the SaaS pricing model. During the implementation, various customization requirements have been identified. Among them is the need for a Product Configurator to simplify and accelerate offer generation and to automate sales engineering. This tool is going to be exposed to resellers who can enter sales orders for a defined set of products on their own. The work and the material for entered sales orders are thereafter disposed automatically. The ERP system which is about to be implemented, does not offer such a feature by default. Therefore, it is subject to customization.

In the described case, the customization covers the introduction of a completely new BO which stores the configuration and the extension of the BO *Lineitem* (which is attached to BO *Opportunity*) by a reference to this new business object. Therefore, the new BO *Configuration* is created and attached to a Lineitem. The folding facades produced by the company have characteristics such as height, width, color, and a glass type. All these attributes are persisted. Furthermore, a method to calculate the price of the current configuration is required. The new object structure is visualized in Figure 2 using the Fundamental Modeling Concepts (FMC) (Knoepfel, Groene, Tabeling, 2005).

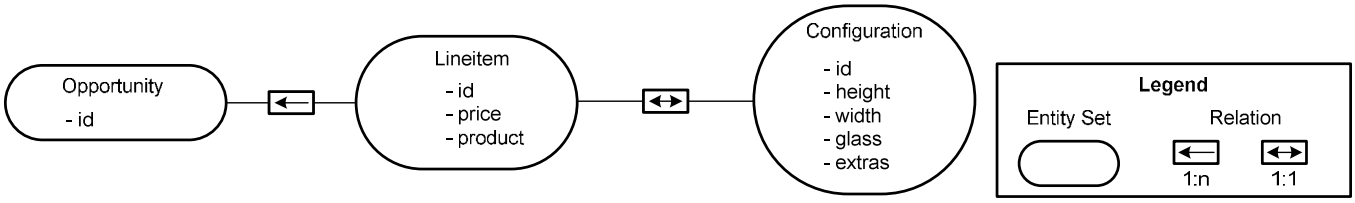

Figure 2. Modifications to the domain model (FMC Entity-Relationship Diagram)

In Mueller et al. (2009), we described how to modify the domain model to meet the requirements of the customer. Furthermore, the *Configurator* UI component must be able to remotely consume the additional data and trigger custom business logic such as the calculation of the price for the current configuration (see Figure 3).

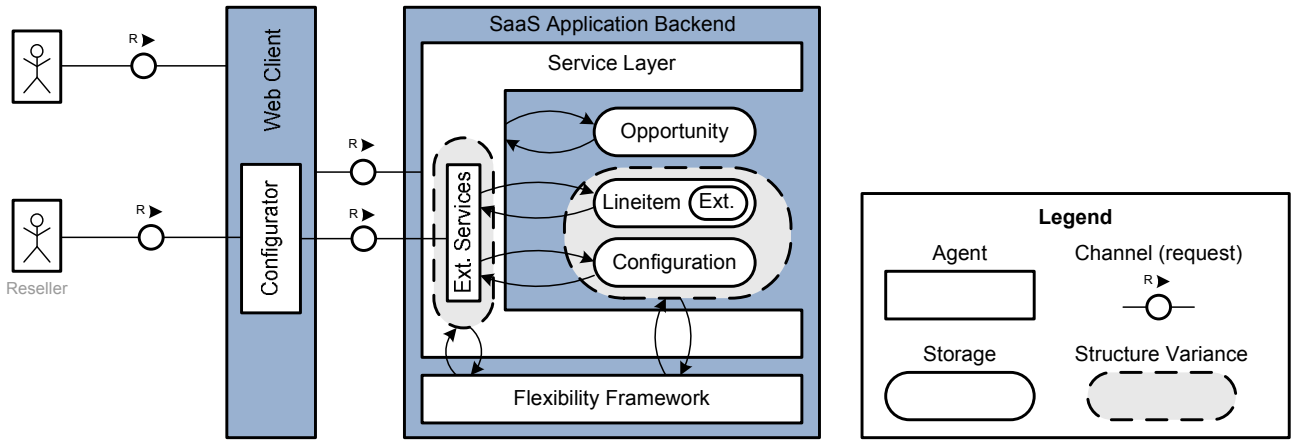

Figure 3. Architecture with extended Service Layer (FMC Block Diagram)

The extension of BO *Lineitem* and the new BO *Configuration* require additional services to be consumed by the *Configurator* component. A Flexibility Framework, which is not in the scope of this paper, has to integrate the changes into the existing domain model and must expose the extended features as services.

Web Service Technologies

In the following, we will introduce technologies and architecture styles that facilitate the implementation of a service layer.

SOAP

SOAP is a network protocol that defines an XML-based messaging framework that can be used to "exchange structured and typed information between peers in a decentralized, distributed environment" (Mitra and Lafon 2007). SOAP acts as a standardized envelope for data elements that are defined in a common, interoperable format (Fremantle, Weerawarana, Khalaf, 2002). SOAP furthermore supports the definition of bindings to underlying protocols to transport these envelopes as messages.

REST

Representational State Transfer (REST) is a software architecture style for distributed hypermedia system that was first introduced by Fielding (2000). The principle element in REST is a *resource*, any piece of information such as a picture, hypertext, or a BO. Besides raw data, every resource contains *representation* metadata that describes the resource. Typical examples for representations are documents, files, or messages. A resource can be uniquely identified by a *resource identifier*, for instance a URL. Interaction with resources is by default stateless, requiring each request to contain all necessary information to fulfill the current operation. The communication interface leveraged for the information exchange must be uniform and decoupled from the actual service provided. The most commonly used example for a uniform interface is a binding to the four most frequently used operations of the HTTP protocol: *GET*, *POST*, *PUT*, and *DELETE*.

- *GET* is used to retrieve a representation of the resource referenced by the resource identifier. GET operations are *safe*, meaning that they may not have any impact on the state of the resource on the server side. Consequently, GET operations are *idempotent*, which means that multiple requests will always cause identical side effects (Schulz-Hofen, 2007).
- *POST* operations cause state changes and are typically used to create new resources in the context of the resource referenced by the resource identifier in form of a URI. If a POST request delivers a representation in its message body, the represented resource will be stored and appended as a subordinate of the referenced resource (Schulz-Hofen, 2007). Consequently, POST operations are neither idempotent nor safe.

- *PUT* updates or creates the resource identified by the URI and is consequently idempotent, but not safe.
- *DELETE* operations request to delete the resource specified in the URI. DELETE is not safe, but idempotent.

IMPLEMENTATION

In the following, we will describe our approach for a service layer that copes with the challenges described in the Problem Statement. The resulting implementation will be validated based on the given show case. In our last paper, the implementation of a flexible, multi-tenancy-enabled backend was based on RubyOnRails. Thus, it is recommended to take over this decision and develop the service layer in Ruby and leverage the features of the Rails framework.

Basic Assumptions

A basic requirement for the service layer is the ability to adopt and expose modifications of the underlying domain model in an automated manner without the need for further manual intervention. This domain model "creates a web of interconnected objects" (Fowler and Rice, 2006), which we defined as Business Objects consisting of data and behavior and implemented in Ruby. Consequently, an automated adoption of modifications to a Business Object can only be reached if the relationship between services and BOs is a 1:1 mapping. If every BO is surrounded by its own service, and if service methods can be automatically mapped to methods of the underlying object, it is possible to expose new features through additional service methods. By implication, two dependent BOs expose two web services that depend on each other.

Before we can implement an object-oriented service layer as described above, it is recommended to design the general architecture of the service layer as well as methodologies to describe and access the services available.

A RESTful Interface

Both approaches, SOAP as well as REST, can be used to implement the service layer. Nonetheless, each of them has its specific advantages and drawbacks, as described by Cappell (2009). SOAP focuses on messages and access to named operations, while REST emphasizes the exposure of resources with a hierarchical, URI-based paradigm. Both features will be required, but a resource-based approach is close to our demand for an object-oriented service layer. We will thus draw on REST and expose BOs as resources. Consequently, attributes of a specific BO are referenced by a URL that identifies the BO and points to the subordinate attribute (see Figure 4).

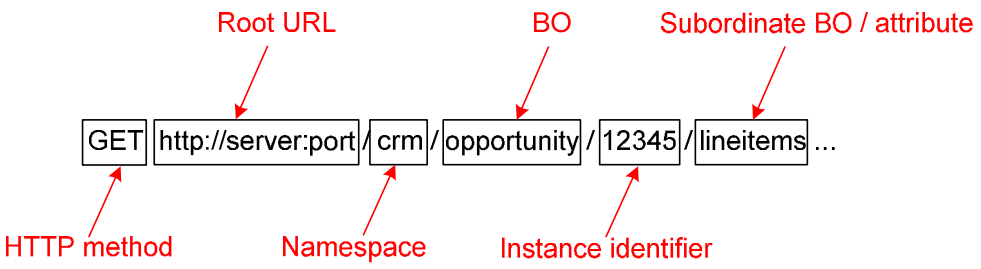

Figure 4. Example REST request to retrieve all *Lineitems* of *Opportunity* "12345"

By implementing a *RESTful* service interface, we are able to structure our domain model in a hierarchy based on namespaces and parent-child relationships. It is furthermore possible to perform CREATE, READ, UPDATE, DELETE (CRUD) operations on each BO. However, our domain model contains object-oriented BOs that consist of data and behavior. Consequently, a structured way to perform BO operations other than CRUD has to be provided.

A popular approach that is followed by service platforms such as Facebook, Amazon S3, or Flickr is to define a REST-like service interface that takes up basic concepts of REST, but offers enhanced functionality. We follow this approach and define a grammar to execute BO-specific operations (see Figure 5).

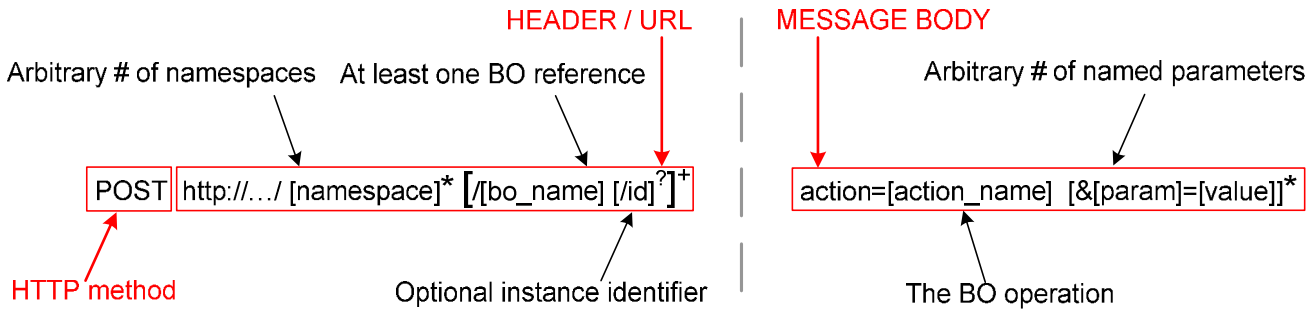

Figure 5. A RESTful interface to facilitate BO-specific operations using HTTP POST

It is now possible to execute operations on the BO (instance) that is referenced by the resource identifier by using a dedicated *action* tag as well as optional method parameters in the POST message body.

We cannot predict the side effects and semantics of specific BO operations, but we must assume that they may change the state of the resource referenced. Thus, calling methods with HTTP GET would violate the *safety* constraint. Calling BO-specific operations can be best compared to an UPDATE operation. Since most browsers are not capable of sending PUT messages, it is recommended to use HTTP POST.

Service Object Descriptors

By implementing a REST-like service as described above, we are now able to create, read, update, and delete elements of the domain model. It is furthermore possible to execute any operation that is offered by the respective Business Object. However, a proper use of the service layer is only possible if sufficient metadata can be provided. Otherwise, the result of operations or the structure of each object would be unknown to the service consumer, which complicates the development of service clients. Thus, we need to provide structured information that sufficiently describes the shape and features of the underlying domain model elements. We will start by defining a new type of object:

Service Objects (SOs) are real-world entities that expose behavior and data through a service layer. Each SO leverages an underlying Business Object that implements the exposed functionality and data. It is described by a *Service Object Descriptor (SO Descriptor)*, including its attributes, operations, and associations to other objects. In Mueller et al. (2009), we introduced a basic Domain-Specific Language (DSL) based on XML to specify BOs. We will follow this approach and define the *Service Object Description Language (SODL),* which consists of the following key elements derived from the underlying BOs:

- *Description Root:* The root element of a SO Descriptor contains the BO class that is referenced by the SO.
- *Associations:* All associations to other objects need are listed under this node. Each association is described by a relationship type / cardinality (“belongs_to”, “has_one”, or “has_many”), the name of the association, as well as the target class and a visibility tag (“public”, “private”).
- *Properties:* All attributes of the underlying BO that shall be exposed are listed here with a description of their type and a visibility tag (“read”, “write”, “readwrite”).
- *Methods:* This node contains all BO methods the Service Object may expose to consumers. Each method is defined by a name, a return type, and an optional list of parameters.

An example of a SO Descriptor for the object "Configuration" is visualized in Listing 1.

```
<business_object name="Configuration">
  <associations>
    <association cardinality="has_one" name="material" target_type="Material" visibility="public"/>
    <association cardinality="has_one" name="glass" target_type="Glass" visibility="public"/>
    <association cardinality="has_one" name="color" target_type="Color" visibility="public"/>
    <association cardinality="belongs_to" name="lineitem" target_type="Lineitem" visibility="public"/>
  </associations>
  <properties>
    <property name="width" access="readwrite" type="Float"/>
    <property name="length" access="readwrite" type="Float"/>
    <property name="weight" access="read" type="Float"/>
    <property name="price" access="read" type="Integer"/>
  </properties>
  <methods>
    <method name="get_progress" return_type="Integer"/>
  </methods>
</business_object>
```

Listing 1. SO Descriptor for *Configuration*

The scope of SODL is similar to WSDL (Christensen, Curbera, Meredith, Weerawarana, 2001), namely to describe the respective web service at design time. However, WSDL defines many elements, such as port types, end points, or bindings, which are not required when using REST. Although these shortcoming have been tackled with WSDL 2.0 (Mandel , 2008), no built-in means to describe BOs (including their attributes, methods, and associations) are provided, but an additional notation on top of WSDL would be required.

The SO Descriptor acts like a *Facade* (Gamma, Helm, Johnson, Vlissides, 1995) and allows a clear separation between the Service Layer and the underlying BO. Only methods, associations, or parameters that are listed in the SO Descriptor may be accessed by external service consumers, while the BO itself may contain additional data or behavior.

Business Object Extensions

In the presented show case, a new Business Object named *Configuration* needs to be implemented and made accessible via services. Furthermore, the existing BO *Lineitem* will be extended by a reference to *Configuration*. Consequently, the BO and the SO Descriptor of *Lineitem* have to be modified. Similar to new BOs, an extension to an existing BO consists of a source code file that introduces new data and behavior and a SO Descriptor with a slightly different schema. The description file for the extension of *Lineitem* is shown in Listing 2.

```
<bo_extension name="LineitemExtension" parent="Lineitem">
  <associations>
    <association name="configuration" cardinality="has_one" target_type="Configuration" … />
  </associations>
  <properties/>
  <methods/>
</bo_extension>
```

Listing 2. Service Object Descriptor for *LineitemExtension*

In the given example, the extension of *Lineitem* adds a single association to the BO Configuration. The SO Descriptor furthermore contains a reference to the core BO *Lineitem* as well as the name of the extension.

Once the customer of the extension attempts to install the extension, the Flexibility Framework (see Figure 3) will merge both descriptions into a new, tenant-specific SO Descriptor and store this metadata in a repository.

Service Request Handling

The dynamic nature of the implemented SaaS application bears additional challenges to service request handling. It is possible that additional BOs may have to be loaded into the application at run time. This implies that a service request for a Service Object may arrive at a time where the underlying BO class as well as the SO Descriptor are not yet loaded and available to the application. However, some entity has to handle the request in the first place. Thus, a *Generic Controller* by default receives all requests, retrieves the tenant-specific context information consisting of the SO Descriptor and the BO class, and forwards the request to the specific BO (see Figure 6).

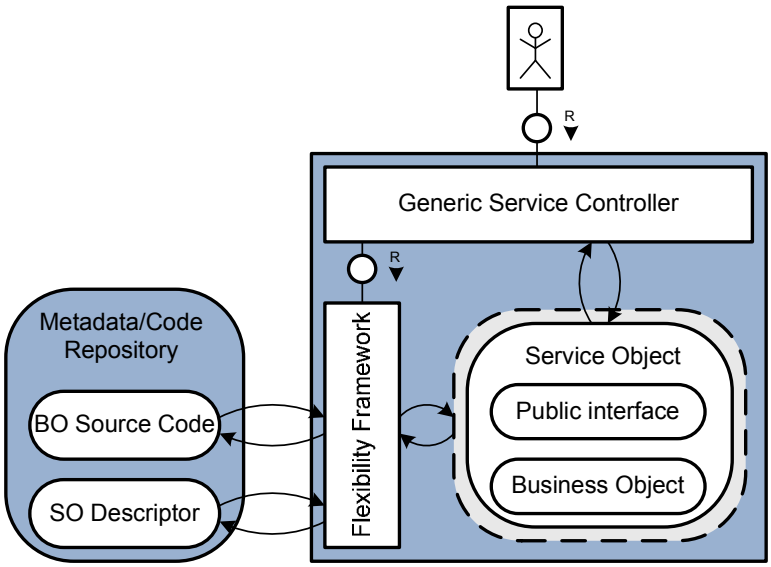

Figure 6. General Architecture to handle service requests (FMC Block Diagram)

The Generic Controller has to perform certain tasks before it forwards the requests to the BO. The most important of these tasks are:

- *Authentication and authorization:* Once the identity of the requesting user has been proven, the user-/tenant-specific context, consisting of the SO Descriptor and the tenant-specific Business Object, can be loaded. Based on the metadata description, the Generic Controller now consumes the Public Interface to decide whether the user is allowed to execute the requested operation for the specific BO or not. This interface is generated by the Flexibility Framework and consists of the methods, attributes, and associations listed in the SO Descriptor.

- *Resolving requests:* In order to be able to forward the request to the BO, the Generic Controller has to identify which BO was actually referenced by the request. Following the resource orientation of the REST paradigm, this information can be retrieved by analyzing the request URL. If the requested action was meant to be performed on a BO instance, the Generic Controller will load this instance and hand over the request to the BO instance instead of the BO class. A "create" request for an Opportunity for example should be handled by the BO class *Opportunity*, and not by an already existing *Opportunity* instance. Updating the status field of the BO *Opportunity* from "Open" to "Closed" on the other hand only makes sense on an instance of *Opportunity*.

Before the request can be handed over to a BO class or instance, it is necessary to map the request action to a BO operation and to bring all parameters in correct order. This step is only important in case of web service requests, because parameters defined in a URL may occur in a different order than expected, leading to a BO method call with corrupted data. Thus, the SO Descriptor is used identify the target method and to reconstruct the correct order of parameters (see Listing 3).

```
1   def call_matching_method(obj, method_name, params)
2     # raise exception if the method does not exist
3     unless obj.service_methods.include? method_name.downcase
4       raise ArgumentError, "Method '#{method_name}'' does not exist!"
5     end
6     # raise exception if parameter count is not equal
7     unless obj.service_methods[method_name.downcase].size == params.size
8       raise ArgumentError, "Method '#{method_name}'' requires a different number of parameters!"
9     end
10    # check if all expected parameters are delivered
11    plist = []
12    obj.service_methods[method_name.downcase].each do |p, type|
13      unless params.include? p
          raise ArgumentError, "Method '#{method_name}'' requires different parameters!"
14      end
15      plist << params[p]
16    end
17    # call the method
18    obj.send(method_name, *plist)
19  end
```

Listing 3. Implementation of a matching algorithm from Service operations to BO methods

A service method and a BO operation can be considered to match, if their names (line 3) as well as their parameter count (Line 7) and parameter names (Line 13) are equal.

Once the BO operation has been identified and all parameters are in correct order, the BO will execute the operation (Line 18). Finally, the results (if there are any) are returned to the Generic Controller, who serializes the result object to XML or JSON and sends a response containing the serialized object back to the requesting party.

VALIDATION

In the Problem Statement, we depicted several challenges regarding the construction of a services layer in a multi-tenant enterprise application. In this section, we will validate the concepts described in this paper against these issues and we will furthermore evaluate how our implementation matches the needs described in the show case.

- **Implement a service layer to expose the domain model:** We introduced the concept of an object-oriented service layer that provides services for all elements of the domain model. This service layer follows a REST-like approach and can be consumed via basic HTTP methods.
- **Follow a model-driven approach to enable the immediate and implicit adaptation of changes to the domain model(s):** We introduced *Service Objects* that expose the features of underlying BO according to a SO Descriptor. The interpretation of this model described in metadata as well as the construction of the SO takes place at runtime when a request for the specific object is received. Consequently, modifications of the domain model will be immediately and automatically adapted by the surrounding service.
- **Clearly separate all tenants from each other and provide unique services that match the domain model of the specific tenant:** The *Generic Controller* is able to identify the appropriate tenant based on the login information and will provide this information when it retrieves the required objects and metadata. Neither the SO Descriptor, nor any third party source code is able to reference entities outside the own tenant, since the multi-tenancy is completely encapsulated by the framework.

In the show case presented, a customer organization required additional Business Objects as well as BO extensions in order to automate their sales processing. We have provided a dynamic and flexible service layer that is capable of exposing custom business logic by drawing on metadata. We have furthermore described the shape of the additional metadata required in the given showcase (see Listings 1 and 2). Given a flexible domain model as described in Mueller et al. (2009), we are able to integrate this custom business logic and automatically adapt the tenant-specific service layer without affecting other

customers of the SaaS enterprise application. Consequently, a development partner is now able to implement a *Configurator* UI component that accesses these additional services and facilitates the automation of the sales processing as described in the show case.

CONCLUSION

The success of a SaaS enterprise application highly depends on the cost-saving potential, in contrast to a traditional enterprise application. The new business model ultimately forces SaaS vendors to implement a multi-tenant architecture to make their product affordable for SMEs.

Despite a shared infrastructure, the solution has to be flexible enough to facilitate tenant-specific domain models. Consequently, the services that expose this domain model have to be tailored to the specific requirements of each customer organization without affecting other tenants or the common code base.

In our paper we provided a lean, interpreted approach to implement a service layer that fulfills these requirements. We furthermore introduced a real-world show case that covers the implementation of a typical partner extension. Based on the architecture proposed, we provided means on how a partner could extend the core business logic to satisfy the requirements of the customer organization. Finally, we validated our approach by drawing on the requirements of a real-world show case.

By providing a multi-tenant, but flexible enterprise architecture with tenant-specific domain models and a service layer that is automatically derived from this model, we aim to encourage 3rd party developers to enhance the basic application and frame a rich ecosystem of partners.

Outlook

The model-driven service layer we proposed in this paper enables service consumers to request SO Descriptors in order to learn about the structure of the Service Objects available. However, to make our solution usable with low implementation effort, we need to provide tools to create client side object proxies, similar to a WSDL Stub Generator.

In the limited scope of this paper, we furthermore assumed that all services of an enterprise application can be bound to specific BOs, leaving out cross functionality or higher-level operations. It is likely that a SaaS ERP solution will require an additional set of services. We will conduct further research on the implementation of a *Composite Layer* on top of the architecture presented in this paper.

REFERENCES

1. Mertz, S., Eschinger, C., Eid, T., Pring, B. (2007) Dataquest Insight: SaaS Demand Set to Outpace Enterprise Application Software Market Growth, *Gartner Dataquest*
2. Fink, L., Markovich, S. (2008) Generic Verticalization Strategies in Enterprise System Markets: An Exploratory Frameworkm, Beer-Sheva, Israel, Ben Gurion University of the Negev
3. Chong F., Carraro G. (2006) Architecture Strategies for Catching the Long Tail, *MSDN Library,* Microsoft Corporation, *http://msdn.microsoft.com/en-us/library/aa479069.aspx*
4. Bennett, K., Layzell, P., Budgen, D., Brereton, P., Macaulay, L., Munro, M. (1999) Service-Based Software: The Future for Flexible Software, Durham, UK, University of Durham, http://www.bds.ie/Pdf/ServiceOriented1.pdf
5. Sääksjärvi, M., Lassila, A., Nordström, H. (2007) Evaluation The Software As A Service Business Model: From CPU Time-sharing to Online Innovation Sharing, *Proceedings of the sixth conference on IASTED International Conference Web-Based Education – Volume 2*, ACTA Press, Anaheim, CA, California, pages 322-330.
6. Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. (2007), A framework for native multi-tenancy application development and management, *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, CEC/EEE 2007, The 9th IEEE International Conference on, pages 551-558, July 2007.
7. Mueller, J., Krueger, J., Enderlein, S., Helmich, M., Zeier, A. (2009) Customizing Enterprise Software as a Service Applications: Backend Extension in a Multi-tenancy Environment, *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS 2009)* (to appear), May 6 – 10, Milan, Italy, http://epic.hpi.uni-potsdam.de/pub/Home/JuergenMueller/Mueller_Krueger_Enderlein_Helmich_Zeier_-_Customizing_Enterprise_SaaS_Applications.pdf

8. Schulz-Hofen, J. (2007) WebData - Definition of a Middleware for Exposing and Accessing Object-oriented Domain Models as Web, September 25, Hasso-Plattner-Institute for IT-Systems Engineering, Potsdam, Germany, http://www.scribd.com/doc/2218300/WebData-Defnition-of-a-Middleware-for-Exposing-and-Accessing-Objectoriented-Domain-Models-as-Web-Resources
9. Wirdemann, R., Baustert, T. (2007) RESTful Rails Development, March 26, b-Simple, Hamburg, Germany
10. Fowler, M., Rice, D. (2006) Patterns of the Enterprise Application Architecture, Addison-Wesley, Boston, MA, USA
11. Knoepfel, A., Groene, B., Tabeling, P. (2005) Fundamental Modeling Concepts: Effective Communication of IT Systems, Wiley, Chichester, England
12. Mitra, N., Lafon, Y. (2007) SOAP Version 1.2 Primer, *W3C,* http://www.w3.org/TR/soap12/.
13. Fremantle, P., Weerawarana, S., Khalaf, R. (2002) Enterprise Services, Examining the emerging field of Web Services and how it is integrated into existing enterprise infrastructures, *Communications of the ACM*, October 2002/Vol. 45 No. 10, 77 – 82
14. Chappell, D. (2009) SOAP vs. REST: Complements or Competitors?, *2009 ESRI Developer Summit Keynote*, March 25, http://www.esri.com/events/devsummit/pdfs/keynote_chappell.pdf
15. Mandel, L. (2008), Describe REST Web services with WSDL 2.0, IBM developerWorks library, May 29, http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/
16. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. (2001) Web Services Description Language (WSDL) 1.1, *W3C Note,* http://www.w3.org/TR/wsdl
17. Fielding R. (2000) Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, USA, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam, 1st Edition, March 1995