An Architectural Model for Customizing the Business Logic of SaaS Applications

André Correia, Jorge Renato Penha and António Miguel Rosado da Cruz
Escola Superior de Tecnologia e Gestão, Instituto Politécnico de Viana do Castelo,
Av. Do Atlântico s/n, 4900-348, Viana do Castelo, Portugal

Keywords: Multi-tenancy, Software as a Service, Saas, Business Logic Configurability, Customizability, Extensibility.

Abstract: Traditional software applications are typically customized before being delivered to a client. This customization was a paid service delivered by software development organisations. With the growing demand of applications delivered with a SaaS model, software development organisations are increasingly responding with the migration of traditional applications to a multi-tenant SaaS deployment model. This makes them face themselves with the problem of customizing a shared application, with a shared database, for each tenant that subscribes their deployed service. After overviewing existing solutions for application customizability, this paper addresses the customization of the business logic layer of multi-tenant applications by proposing a solution, which has been used in a multi-tenant WMS application deployed with a SaaS service model.

1 INTRODUCTION

Traditional software development firms commonly develop software applications that are customized, either by themselves or by affiliated companies, before being deployed in their clients' locations. Their business is about developing software as much as customizing that software to each specific client. This supports the fact that applications need to be flexible to a certain point that allows them to accommodate variability in the response to the customer's requirements (Gebauer and Schober, 2006).

These software development firms are increasingly facing the challenge of having to adapt their applications for deploying them in the cloud with a software-as-a-service (SaaS) delivery model.

The SaaS model provides a multi-tenant, ready to run, on-demand hosted application. Multi-tenancy is, indeed, the primary characteristic of SaaS applications, as it allows the service provider to run a single instance application, which supports multiple tenants on the same platform. This involves sharing unique resources, as a database and an application instance, giving the tenants' users the impression that they are the only ones using those resources. This implies addressing many issues, in order to assure the functional and non-functional isolation of the tenants (Krebs et al., 2012).

Other desirable feature of SaaS applications is that they retain the ability to be customizable. This ability shall not, by any means, threaten the imperative of tenants' isolation.

There are several levels of application customizability, from simple configuration at allowed application points, to tenant specific code extensions at any point of the application, passing by simple extensions to the data model. Also, this customization ability may be the application provider's responsibility or the tenant's responsibility. Either way, a tenant's customization may not interfere with other tenants' application usage experience, even when the customization is the provider's responsibility.

After a survey of existing solutions for application customizability, this paper proposes an approach for functionality customization per tenant, by recurring to specific code extensions that may be plugged into specific points in the application. The approach is being used in a warehouse management system (WMS) application that will be deployed with an SaaS service model. The structure of the presentation is as follows: the next section discusses the customizability of software applications, explains why traditional approaches are not suitable for multi-tenant SaaS applications, and presents

Correia A., Renato Penha J. and Miguel Rosado da Cruz A..
An Architectural Model for Customizing the Business Logic of SaaS Applications.
DOI: 10.5220/0004490401620168
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 162-168
ISBN: 978-989-8565-68-6

related work framing it in three architectural levels, namely the data, presentation and business logic layers; section 3 presents the general view of our approach for per-tenant customization at the business logic level; section 4 details the approach; section 5 discusses our approach and compares it with existing approaches; and, section 6 concludes the paper and proposes some future research directions.

2 APPLICATION CUSTOMIZATION

Traditional applications, meaning single tenant applications that are deployed on premises, regardless of being web oriented or not, are typically customized by the application provider, or deploying organization, having its data model extended or adapted to the customer's reality, and/or its code modified or extended to meet the customer's business rules. These customization procedures can be made by exploring configurability capabilities of the application, which is the common approach in Software Product Lines Engineering (SPLE), where customized product variants may be derived from a feature model that includes predictable features' variability modelling (Clemens and Northrop, 2001). Or it can be made by changing the application's source code, and culminate with the creation of a new, different, application variant tailored for the new customer/tenant, which has been a common approach for software houses representing and reselling software from major companies but making customized deployments of those softwares.

Gebauer et al. (2006) identifies two types of software application flexibility: **flexibility-to-use**, regarding the features that are provided at the time of deployment, and **flexibility-to-change** regarding the features that constitute an option for later system change.

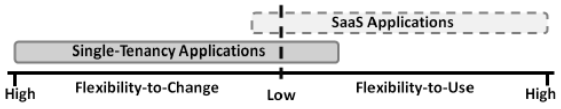

Figure 1: Addressing the two types of flexibility (taken from Ruehl et al., 2011).

Adherence to each type of application flexibility differs according to being a single tenant application or a multi-tenant one (see Figure 1). Single tenant, traditional, applications' customization is typically addressed before deploying the application in the customer/tenant's location, and so they require flexibility-to-change, that is flexibility for changing the features to adapt a software application to a specific customer's requirements, even if the application must be shut down for a period of time. This is also the kind of flexibility addressed by SPLE. These application tailoring procedures are not applicable to the customization of multi-tenant SaaS applications, which, on the other hand, do not require high flexibility-to-change, but do require high flexibility-to-use, meaning that the deployed features must be easily changeable, without affecting the application usage (Ruehl and Andelfinger, 2011).

We consider application customizations at three architectural levels:

- Data level customization;
- Presentation customization;
- Business logic customization.

2.1 Data Level Customization

At the data level, a customizable application typically enables the creation of new entity attributes for the existing entities or, less often, it may even enable the creation of new entity types.

Common data extension customization approaches are (Chong et al., 2006):

- Preallocated fields;
- Name-value pairs;
- Custom columns.

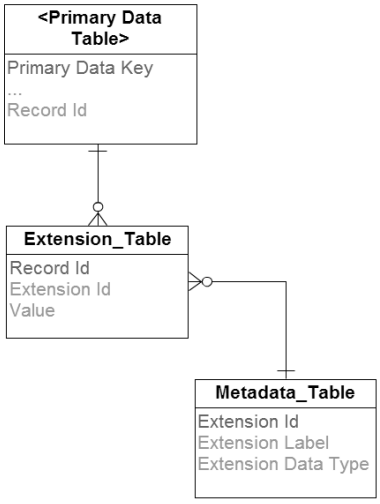

Figure 2: Name-value pairs with extension tables (adapted from Chong et al., 2006).

Preallocated fields are extra fields (columns) that are created in the extendable entities (database tables),

and that may have a different meaning depending on each customer/tenant's will. The number of customizable extendable fields is predetermined in each data table.

Name-value pairs allow the definition of an arbitrary number of extended fields.

Typically, this is enabled by providing the application with a metadata table, defining the extended field (its name or label and its data type), and an extension table, defining the field value and associating it to a field in a primary data table (see Figure 2).

Custom columns are a data extension approach where columns are arbitrarily added to specific tables by making the software dynamically use data definition language (DDL) operations in the database.

Whichever method is chosen to extend the data model, it must be combined with the necessary code adaptation, either by directly modifying the source code, or by providing a mechanism for integrating the additional fields into the application's functionality.

In multi-tenant SaaS single instance applications, the most suitable solution seems to be name-value pairs, because it does not limit the number of extra fields by table nor requires DDL operations in the shared multi-tenant single instance database.

In a multi-tenant name-value pairs approach, the metadata table must be bound to the tenant Id (Chong et al., 2006). And the software code that uses it, must take the tenant into account, without interfering with the other tenants.

2.2 Presentation Customization

Another common kind of application customization is at the presentation, or user interface, level. The customer naturally wants the application to be aligned with the company's corporate image, and its country culture/localisation (language, currency and other cultural peculiarities).

In multi-tenant applications this must also be customizable for each tenant, without interfering with the other tenants' application usage experience.

2.3 Business Logic Customization

After exploring the variability incorporated into the application's features, customization at the business logic level requires that the business logic code, which is typically located at an application layer or in the database layer, is adapted to the customer.

In single tenant applications this is commonly accomplished by modifying the application's source code in order to adapt it to the customer's specific requirements, which could not be foreseen when designing the application flexibility that addresses the variability points.

However, in multi-tenant applications this is not a suitable solution. Modifying the source code is out of question, because it would create a jumbled mix of different tenants' business rules into the source code. Additionally, it would be needed to shut down the (multi-tenant) system every time a tenant would want a piece of customized code.

One of the first successful SaaS applications to appear in the market was Salesforce's CRM solution. Salesforce offers two business logic customization approaches: point-and-click configuration, and code based customization. The former enables fast and easy customizations, by providing a series of simple point-and-click wizards with limited customization capability. And, the latter is useful for deeper customizations to meet more demanding tenants' needs, and is made possible through a native programming language called Apex for tenants to customize complex business logic (Salesforce, 2013); (Weissman and Bobrowski, 2009); (Chen et al., 2010).

Other authors have proposed customization approaches for multi-tenant SaaS applications. For instance, Yaish et al. (2012) propose a conceptual architecture design using elastic extension tables and a number of database, user interface and access control services, for customizing the data layer, the user interface layer and the access control, but it doesn't address business logic customization.

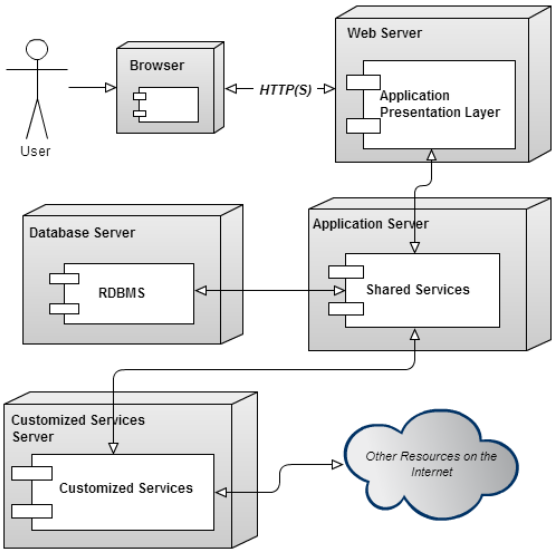

Figure 3: Architecture of the SaaS customizable multi-tenant application.

Xiuwei et al. (2012) propose a business rule engine-based framework for customizing the business logic layer of multi-tenant SaaS applications. Their approach separates the business rules, defined in decision tables, from the software source code, enabling its customization by the tenants within the variability scope pre-determined in the decision tables.

Chen et al. (2010) propose an approach to business logic SaaS applications' customization based on domain engineering techniques and business rules templates. Like Xiuwei's approach, it enables the customization of business rules within the variability scope pre-determined in the rules templates.

3 BUSINESS LOGIC CUSTOMIZATION OF SAAS APPLICATIONS – GENERAL VIEW

Our approach to the customization at the business logic level aims to enable the SaaS provider organisation to be able to supply, as a paid service, the customization of the SaaS application to specific tenants. Note that all the predictable variability in requirements shall be incorporated into the application, leaving to this approach only the unforeseen deeper customization needs.

Figure 3 depicts the architecture of the proposed solution, consisting in a customizable multi-tenant application provided with a SaaS deployment model. The approach requires that customized web services are developed for a given tenant, and that the system is configured, for that tenant, by using a configuration tool, as further explained below.

The user accesses the application's presentation layer, which calls the shared services. These are a set of multi-tenant enabled services that, in turn, access the single instance database.

For customizing the SaaS application business logic, customized services must be put available in a customized services server, or any other web server, and the SaaS application must be configured to plug those services in the desired extension points available in the application.

Let's analyse, through an example, the proposed approach. Consider that a given tenant wants to modify the default behaviour of an order registering SaaS application so that, when a user inserts a new order, the total accumulated debt of the client be verified and, if that debt is above some threshold, the system rejects the new order.

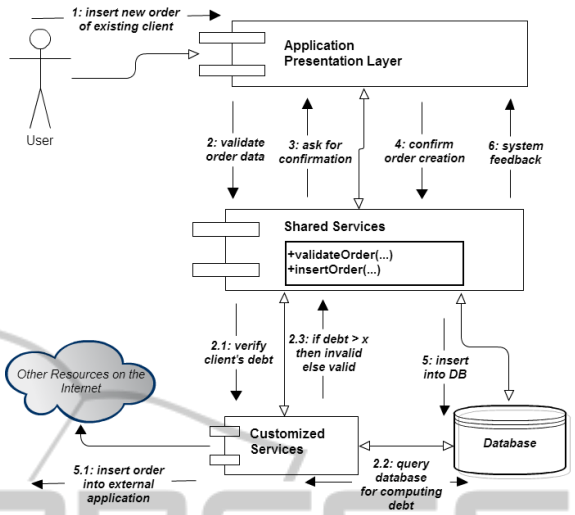

Figure 4: Illustration of an example of the SaaS customizable multi-tenant application.

Suppose, also, that this could have not been foreseen at design time, and incorporated in the application as a feature variability point, as recommended by SPLE. This way, the system needs to be configured so that, when validating the order form, it invokes a customized service that verifies the client's total debt. Figure 4 illustrates this example: first, the user inserts the new order data in the appropriate form (step 1, in the figure); then, the system validates the form, with the shared business logic (step 2). For any other tenant, step 3 would follow, with the system asking for confirmation, but, for the tenant in question, an additional validation is made, by invoking a customized service for verifying the client's debt (step 2.1). This service accesses the database (step 2.2), or any external data (e.g.: from another application), and validates or invalidates the order (step 2.3).

Let's consider that, for instance, the same or another tenant would want that, any new order registered in the SaaS application would also be inserted in another external application (e.g.: CRM). Figure 4 illustrates this as step 5.1 that, attached to the insertOrder service, would enable the integration with an external application.

4 PROPOSED APPROACH

4.1 Detailed Approach

Each specific tenant customized service must be

plugged into an application extension point. Although predefined, these extension points allow to plug a customized service into almost every desired point in the application. This is made possible by providing extension points before, instead and after every shared service associated to application forms, including CRUD operations. For supporting this approach, a set of metadata tables has been established (see Figure 5).

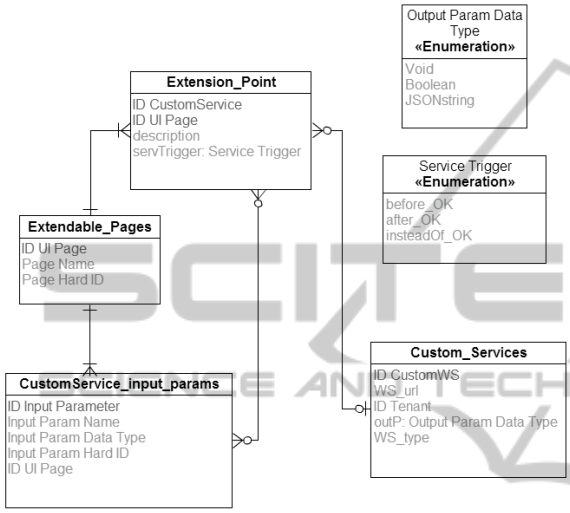

Figure 5: Metadata tables for supporting business logic customization by tenant.

Every pluggable component, provided by an external custom-service, must be registered in table `Custom_Service`, and may have one of three purposes, or types (property `type` in table `Custom_Service`):

- Validate a form field (type: *Validation*);
- Provide data to an external application (type: *Export*);
- Get data from an external application (type: *Import*).

Besides the service type, its URL is also required, just as its result (output parameter) type, and what tenant owns it. The currently allowed result types are:

- *FormValResult*. Form Validation Result, which is composed of a a Boolean, stating if the form is valid, and a String, with a message, in case of invalid form data.
- *Boolean*;
- *Void*, or no result expected. Void and Boolean may be used, for instance, in providing data to an external application.
- *JSON String*. A JSON formatted string that may be used when getting data from an external application (type: *Import*), to show information to the user. Currently, this has the sole effect of opening a dialog box showing the "imported" data.

Note that, regardless of its type, a custom-service may access the application database, through the CRUD shared services. By this way, it can, for instance, import data to the SaaS application from an external source.

Table `Extendable_Page` registers the extendable pages of the application, that is, pages with extension points. Each extendable page of the application's presentation layer may have an extension point, where a custom-service may be plugged in.

A page's extension points are defined in table `Extension_Point`, which also links the extension point to the, possibly Null, custom-service to be called.

An extension point is located around the load and submit operations of an extendable page, and defines the moment when the custom-service, is triggered. The page controller, that is its submit operation handler, handles all the possible operations provided by that page, which may involve the creation of new information (create one or more records in the database) or the modification of existing information (update one or more records in the database).

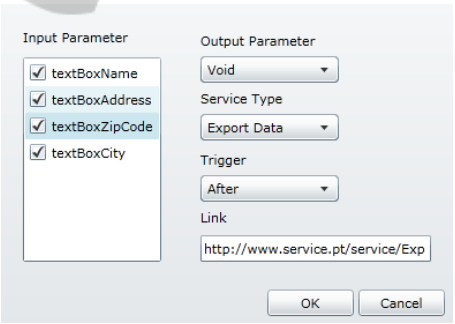

Figure 6: Customization (metadata creation) tool example.

Extendable pages' form fields are identified in table `CustomService_input_params` and may be, then, associated to extension points, gaining the role of input parameters to the plugged custom-service.

Having developed the desired custom-services, the SaaS providing organisation, or the tenant if this feature is given for his/her direct use, may customize the application by using a customization tool (see Figure 6), which dynamically adapts to the extendable page where a custom-service is to be plugged in. This customization tool allows plugging the desired service into an extension point, linking the selected form fields to the web-service's input

parameters, setting the trigger to the appropriate value (*before*, *after* or *instead_of*), and choosing the service type and the output parameter type.

Every extendable page controller has code for looking for custom-services plugged into it, associated to the tenant accessing the page. That is, each extendable page searches for extension points with non-null `ID_CustomService` attached to it, that belong to the tenant accessing the page, in each of the possible triggering positions.

4.2 Validation

The proposed approach to the customization at the business logic level has been tested, and is being used in the development of a WMS application that will be deployed with a SaaS deployment model.

In the WMS application, the user accesses the application's presentation layer through any browser with a Silverlight plugin. The presentation layer calls the WMS domain (shared) services, which are a set of Windows Communication Foundation Rich Internet Application services (WCF RIA services, see for instance http://msdn.microsoft.com/en-us/library/ee707344(v=vs.91).aspx) exposed as SOAP/WSDL. These, in turn, access the WMS database.

For customizing the WMS application business logic, custom REST web-services must be put available in another web server, and the WMS application must be configured to plug those services in the desired extension points, available in the application.

After being deployed, we will further assess the utility and usability of this approach with real customers/tenants and real users in an industry setting.

5 DISCUSSION

The proposed approach enables the tenant-based customization of SaaS applications' business logic. It addresses customizations that could not have been foreseen in a domain engineering analysis, and could not be implemented as a feature variability point as defended by SPLE (Clemens and Northrop, 2001).

The proposed approach makes use of common knowledge technology, in what respects to developers, since the custom-services may be developed in any programming language and may be deployed in any web-server. The only limitation, in the experiences made, and in the WMS application being developed, is that the custom services communicate through REST and that the objects are passed to and from the services with JSON format, because this is what the SaaS application is expecting.

Table 1: Surveyed approaches to SaaS applications' business logic customization.

	Pre-determined variability scope	Full business logic customization	Comments
SPLE	✓		
Sales-force	✓	✓	
Yaish *et al.* (2012)			Addresses only data and presentation layers' customizability
Xiuwei *et al.* (2012)	✓		
Chen *et al.* (2010)	✓		
Our Approach	Addressed through customization of variable features as recommended by SPLE. (Not in the scope of this paper)	✓	

Table 1 aims to compare our approach and the state of the art approaches, referenced in section 2.3, by classifying them according to two main aspects: approaches that only address pre-determined variability scope; and, approaches that enable a full business logic customization. In the first category, we can find the feature variability modelling and product variants of SPLE (Clemens *et al.*, 2001), the point-and-click customization feature of Salesforce (Salesforce, 2013), and the approaches by Xiuwei *et al.* (2012) and Chen et al., 2010).

The approach by Yaish et al. (2012) only addresses data layer and presentation layer customizability. It doesn't address business logic customization at any degree.

As said before, our approach addresses customizations that could not have been foreseen in a domain engineering analysis, and so are outside the limitations of a metadata framework. This way, it assumes that pre-determined feature variability is handled through SPLE or other appropriate approach, but the focus of our approach is, however, deep unforeseen customizations. This way it is only comparable to the Salesforce code customization feature, and the solution is the same, that is making use of open-ended development environments for the most common programming languages to create the needed functionality. In addition, Salesforce also allows creating new functionality using its own proprietary language, Apex.

6 CONCLUSIONS AND FUTURE WORK

Multi-tenant SaaS applications' customization is hard to address because of the requirement for high flexibility-to-use, meaning that the application's deployed features must be easily changeable by one tenant, without affecting the application usage of other tenants.

This paper presented an approach for the tenant-based customization of SaaS applications, at the business logic architectural layer of the application.

The proposed approach reserves the business logic customization to the SaaS provider organisation. The business logic customization may, then, be supplied as a paid service to the tenants that need it. The proposed architecture allows, however, that the business logic customization responsibility is given to the tenant administrator, provided he/she can develop the needed custom-services.

The approach has been tested and is being applied in a multi-tenant SaaS application.

An issue that needs mitigation has to do with the amount of overhead code needed, in each extendable page, to verify if there is any custom-service to be called.

Other future directions involve also the customization responsibility passage to the tenant.

REFERENCES

Chen, W., Shen, B. and Qi, Z. (2010). Template-based Business Logic Customization for SaaS Applications. 2010 IEEE International Conference on Progress in Informatics and Computing (PIC), vol.1, Pages 584-588, 10-12 Dec. 2010, doi: 10.1109/ PIC.2010.5687477.

Chong, F., Carraro, G. and Wolter, R. (2006). Multi-Tenant Data Architecture. MSDN, Microsoft Corporation. Available at http://msdn.microsoft.com/ en-us/library/aa479086.aspx#mlttntda_topic2 (visited in 13th Nov. 2012).

Clemens, P. and Northrop, L. (2001). Software Product Lines: Practices and Patterns. SEI Series in Software Engineering series, Addison-Wesley Professional.

Gebauer, J. and Schober, F. (2006). Information System Flexibility and the Cost Efficiency of Business Processes. Journal of the Association for Information Systems, volume 7, issue 3, pages 122–147, 2006.

Krebs, R., Momm, C., and Kounev, S. 2012. Architectural concerns in multi-tenant SaaS applications. In CLOSER 2012*, 2nd International Conference on Cloud Computing and Services Science*. Pages 426-431, Porto, Portugal, SciTePress 2012.

Ruehl, S. T. and Andelfinger, U., 2011. Applying Software Product Lines to create Customizable Software-as-a-Service Applications. In SPLC'11, *Software Product Line Conference*, August 21-26, Munich, Germany.

Salesforce (2013). http://www.salesforce.com/platform/ customization/ (visited in 15th Feb. 2013).

Weissman, C.D. and Bobrowski, S. (2009). The Design of the Force.com Multitenant Internet Application Development Platform, SIGMOD'09, June 29th-July 2nd, 2009.

Xiuwei, Z., Keqing, H., Jian, W., Chong, W. and Zheng, Li (2012). Business Rule Engine-based Framework for SaaS Application Development. In CLOSER 2012*, 2nd International Conference on Cloud Computing and Services Science*. Pages 345-354, Porto, Portugal, SciTePress 2012.

Yaish, H., Goyal, M. and Feuerlicht, G. (2012). A Novel Multi-tenant Architecture Design for Software as a Service Applications. In CLOSER 2012*, 2nd International Conference on Cloud Computing and Services Science*. Pages 82-88, Porto, Portugal, SciTePress 2012.