A Service Framework for Multi-tenant Enterprise Application in SaaS Environments

Chun-Feng Liao, Kung Chen and Jiu-Jye Chen
Department of Computer Science, National Chengchi University, Taipei, Taiwan

Keywords: Multi-tenant, Schema-mapping, Universal Table, Tenant-specific Storage, SaaS.

Abstract: In recent years, Software as a service (SaaS), a service model for cloud computing, has received a lot of attention. As designing a multi-tenant enterprise application in SaaS environments is a non-trival task, we propose a service framework to deal with three common issues for designing multi-tenant enterprise SaaS applications: tenant context storage and propagation, schema-mapping, and the integration of ORM framework. A prototype and a sample SaaS application have been implemented to verify the feasibility of our framework. In addition, two tenant-specific virtual applications are constructed to demonstrate multi-tenancy. Finally, we conduct a set of experiments to assess the overheads of making an enterprise application multi-tenant enabled.

1 INTRODUCTION

Over the past few years, a considerable number of studies have been made on cloud computing. Among the service models in cloud computing, SaaS (Software as a Service) is reported to be the most competitive (Momm and Krebs, 2011). To be economically scalable, a SaaS application must leverage resource sharing to a great extent by accommodating different tenants of the application while making it appear to each that they have the application all to themselves. Despite the benefits and popularity of multi-tenant SaaS applications, the approach for implementing such applications is still not well-studied and documented (Koziolek, 2012). Technically speaking, SaaS-level multi-tenancy employs a single application instance to serve multiple tenants. In other word, tenants of a SaaS application are oblivious to the fact that the resources (e.g. CPU time, network bandwidth, and data storage) are shared among tenants. For instance, a SaaS application must implement affinity (how tasks are transparently distributed), persistence (how data are transparently distributed and managed), performance isolation, QoS differentiation, and tenant-specific customization (Krebs et al., 2012). These issues are relatively hard to tackle and require higher expertise.

Several architectural issues need to be addressed when implementing a multi-tenant SaaS application, which can be into two layers, namely, the application layer and the data layer. The core issue of the application layer is how to devise a transparent way to store and to propagate the tenant-specific information (or called tenant contexts). Typically, enterprise applications tend to store tenant contexts in a platform-dependent session implementation. However, if the tenant contexts are stored in this way, then in order to propagating tenant contexts into a business method, either the method signature or the body of the business method must be modified to access the platform-specific session implementation. Both of these approaches involve significant modification of code and made business methods being tightly coupled on platform-specific API.

In addition, traditional ”sticky session” (Galchev et al., 2007) handling mechanisms are also platform-specific and require careful configuration in a clustered environment. As nodes in the cloud environment is usually virtualized, heterogeneous and elastic, it is even harder to devise a platform independent approach for handling sticky sessions. A more transparent way to store and propagate tenant context is thus apparently required.

Meanwhile, although it is generally agreed that the multi-tenant data layer is one of the most important concerns (Fang and Tong, 2011), there is also little investigation on data layer concerns in a multi-tenant application. There are two inter-related issues to be addressed in this layer: schema layout management and tenant-specific schema customization. In the design space of the multi-tenant schema layout management strategy, various alternative approaches form a continuum between the isolated data style and the shared data style (Chong and Carroro, 2011). As pointed out by Aulbach et al., the shared data style provides very good consolidation but lacks schema extensibility (Aulbach et al., 2008). Many shared data style assumes that either each tenant has a dedicated set of tables and have the same schema or all tenants are consolidated in one set of tables but share an identical schema. Among commonly used schema-mapping techniques, Universal Table seems to be a promising shared data style since it is possible to preserve extensibility at the same time. Essentially, a Universal Table is a generic structure that has virtually no schema attached to it. Although it is commonly held that Universal Table layout would incur a large amount of performance overhead, it is the approach adopted by SalesForce.com, which is a successful SaaS vendor best known for its CRM service that supports more than 55,000 tenants (Weissman and Bobrowski, 2009). However, it is not clear how the Froce.com SaaS applications leased by tenants transparently transform the query statements for the logical schema to the ones for the physical schema.

As the above-mentioned design issues, application layer or data layer, are cross-cutting concerns of a multi-tenant application, they should be modularized so that developers are able to implement, deploy, and integrate to such customizations to the applications with minimal additional programming and configuration efforts. A general approach is to devise a middleware-level facility that supports transparently tenant context management, automatic mapping of multiple single-tenant logical schemas to one multi-tenant physical schema in the database, and flexible customization of tenant-specific logical schemas. Hence, we propose a middleware-level service framework that addresses these issues.

Specifically, our objective is to design a service framework that provides: 1) a platform independent tenant context management service that stores and propagates tenant contexts based on the thread-specific storage pattern (Schmidt et al., 1996); 2) a data service that implements multi-tenant Universal Table schema layout; 3) a multi-tenant ORM (object-relational mapping) customization service that enables the customization of tenant-specific domain objects and their mappings to the underlying schema layout. On top of this service framework, we construct a simple SaaS application, ShoppingForce.com, to demonstrate the feasibility of our approach. Finally, we also present results of the performance assessments of this work.

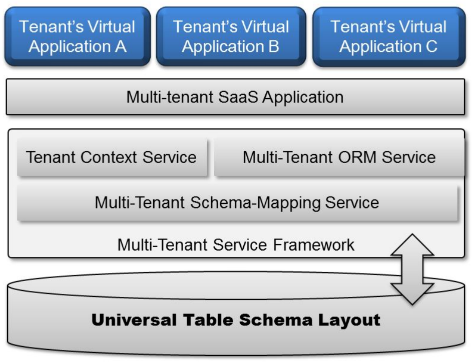

Tenant's Virtual Application A
Tenant's Virtual Application B
Tenant's Virtual Application C
Multi-tenant SaaS Application
Tenant Context Service
Multi-Tenant ORM Service
Multi-Tenant Schema-Mapping Service
Multi-Tenant Service Framework
Universal Table Schema Layout

Figure 1: Overall architecture of the proposed service framework.

2 RELATED WORKS

As mentioned, the design issues of multi-tenant enterprise applications fall into two groups: the application layer and the data layer. In the application layer, several challenges arise when transforming these applications into multi-tenant ones: 1) how to obtain tenant-specific information (or called tenant contexts), 2) where to store the tenant contexts, and 3) how to propagate tenant contexts among components (Bezemer and Zaidman, 2010). Several approaches have been proposed to deal with the first issues mentioned above, including intercepting filters (Cai et al., 2010), aspects (Wang and Zheng, 2010), or contexts (Truyen et al., 2012). What seems to be lacking, however, is an appropriate mechanism for storing and propagating of tenant contexts.

In the data layer, Pereira and Chiueh mentioned the concepts of a multi-tenant query rewriting engine in the future work section (Pereira and Chiueh, 2007) . Li proposes a heuristic-based query rewriting mechanism for transforming queries to the logical schema to HBase, which is an implementation of BigTable (Li, 2010). Aulbach et al. survey several schema-mapping techniques, including Universal Table (Aulbach et al., 2008). The data architecture used by Force.com (Weissman and Bobrowski, 2009) falls into the category of Universal Table, which is the foundation of our data layer design.

Several attempts have been made to provide a common platform for multi-tenant enterprise applications. However, the objectives of these attempts either focus on isolation issues (Azeez et al., 2010), administration issues (Strauch et al., 2012), or only provide conceptual discussion (Shimamura et al., 2010). This paper concentrates the issue of tenant context and data management and realize our approach as a service framework.

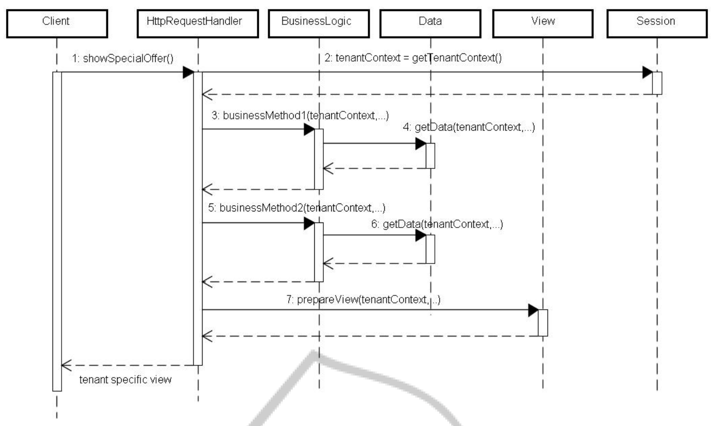

Client
HttpRequestHandler
BusinessLogic
Data
View
Session
1: showSpecialOffer()
2: tenantContext = getTenantContext()
3: businessMethod1(tenantContext,...)
4: getData(tenantContext,...)
5: businessMethod2(tenantContext,...)
6: getData(tenantContext,...)
7: prepareView(tenantContext,...)
tenant specific view

Figure 2: In traditional HTTP Session approach, all method signatures throughout the call sequences have to be modified.

3 DESIGN

In this section, we will present the design of the proposed multi-tenant service framework for enterprise applications. As depicted in Fig.1, The service framework provides three core services that are common for multi-tenant enterprise applications, namely, the Tenant Context Service, the Multi-Tenant ORM Service, and the Multi-Tenant Schema-Mapping Service. A multi-tenant SaaS application, which is able to host several tenant-specific virtual applications, can be built on top of the service framework. The detailed mechanisms of the three core services will be introduced in the following sub-sections.

3.1 Tenant Context Management

A key characteristic of a multi-tenant SaaS application is that it must provide tenant-specific user interfaces, business logics and data to a certain extent. To realize tenant-specific customization of an application, some artifacts, usually called isolation points, have to be isolated for different tenants (Cai et al., 2010). To implement tenant-specific customization, program logics in the isolation points must have access to tenant contexts. Therefore, it is important to find a way for storing and propagating tenant contexts among the isolation points spreading in a multi-tenant SaaS application. If the clients of a SaaS application are Web-based, then a common approach is to store tenant contexts in a platform-dependent HTTP session implementation. Essentially, an HTTP session is an abstraction of a shared storage which is accessible through a specific sequence of user-system interactions. For instance, the shopping cart in an e-commerce web site is usually stored in an HTTP session.

However, the above-mentioned approach leads to several problems. First, the approach is not applicable to non-web-based clients. Second, if not carefully designed, the HTTP session can be unstable and brittle due to the sticky session problem, proxy farm problem, or net quasar problem (Joines et al., 2003). Most importantly, the multi-tenant processing logic will be ”polluted” by platform-specifics of the underlying HTTP implementation. Taking Java-based Web application as an example, the multi-tenant processing logic must use a Servlet API to access tenant contexts, causing the logic being dependent on the Servlet API. Moreover, the method signatures for user interfaces, business logics and data access have to be modified to propagate tenant contexts (see Fig. 2). To minimize the efforts of migrating an single-tenant application into multi-tenant ones, it is preferable to avoid significant modification of code or being tightly coupled with some platform specific API.

Thread-specific storage is a design pattern that allows multiple threads to access a logically global but physically local for each individual thread (Schmidt et al., 1996). Internally, a thread-specific storage is essentially a globally accessible list of maps, where the maps are indexed by thread IDs. Hence, program logics in a specific thread `t` can only access one of the map, that is, the map indexed by `t`. It is reported that thread-specific storage is more efficient, reusable and portable (Schmidt et al., 1996). However, if it is not carefully designed, it can lead to an obscure system structure because of the use of a (logically) global object. As a result, it is important to store and propagate tenant contexts in a thread-specific storage through an uniform API. Based on this observation, we devise a platform independent mechanism that allows the program logic to access tenant contexts, stored in a tenant-specific storage, from user interfaces, busi-

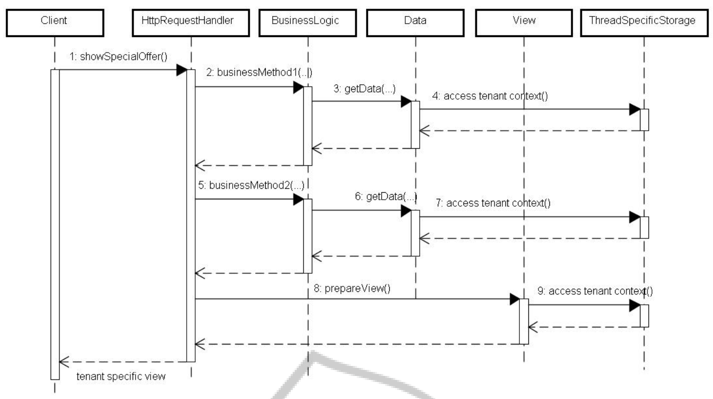


Figure 3: In the tenant specific storage approach, only isolation points are modified.

ness logics and data without depending on the HTTP session, as shown in Fig.3.

3.2 Multi-tenant Schema-mapping

This section describes the design of the proposed Universal-Table-based multi-tenant schema-mapping service. Before examining the detailed mechanisms, it is helpful to introduce the overall data architecture and design issues of Universal Table. Essentially, Universal Table is a generic storage consisting of a GUID (Global Unique Identifier), a tenant ID, and a fixed number of generic data columns (i.e. the *Data* Table in Fig.4). The metadata of logical tables (objects), logical columns (fields), logical relationship, logical primary keys, and logical index information of records are stored in *Objects*, *Fields*, *Relationships*, *Uniquefields*, and *Index*, respectively (see Fig.4). In the sequel, we follow the convention in (Weissman and Bobrowski, 2009) and use the term objects and tables as well as fields and columns interchangeably.

Consider a hypothetical e-commerce SaaS application, ShoppingForce.com, that enables its tenants to sell products and to process orders on-line. Since different tenants have their own unique needs in describing their products, ShoppingForce.com allows its tenants to create their own customized schemas for their products. Fig. 5 illustrates the scenario. Here we have two product tables (i.e. $Product^{l}_{t=667}$ and $Product^{l}_{t=604}$, where l denotes "logical" schema and t denotes tenant id). The data in the two logical tables will be stored together into a universal table (i.e. the *Shared* Table) via the schema mapping service.

We now turn to the design of our Universal-Table-based multi-tenant schema-mapping service. At the core of the service is a set of query rewriting rules that specify the transformations from logical queries to physical queries via relational algebra. Due to space limitation, the reader is referred to our previous

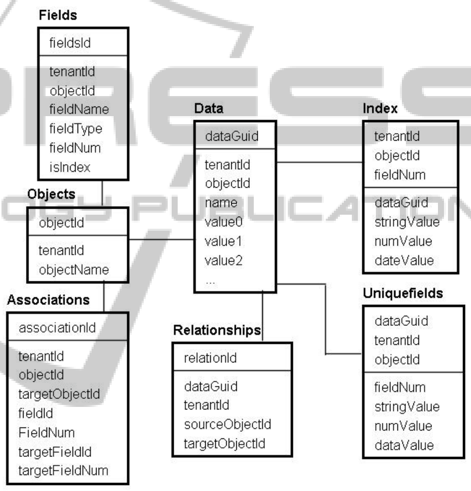
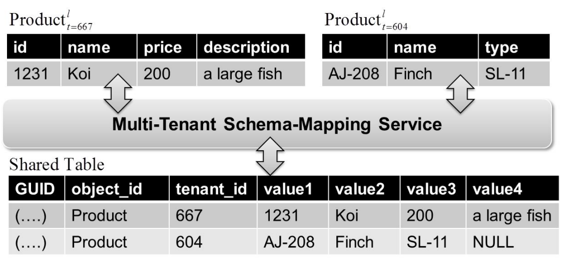


Figure 4: The data architecture of Universal Table schema-mapping.



$Product^{l}_{t=667}$

id	name	price	description
1231	Koi	200	a large fish

$Product^{l}_{t=604}$

id	name	type
AJ-208	Finch	SL-11

Shared Table

GUID	object_id	tenant_id	value1	value2	value3	value4
(….)	Product	667	1231	Koi	200	a large fish
(….)	Product	604	AJ-208	Finch	SL-11	NULL

Figure 5: Example of Universal Table schema-mapping.

work for detailed specification of those query rewriting rules (Liao et al., 2012). In the following, we sketch the overall query rewriting mechanism via an example of transforming a projection statement. Let us assume that a tenant, whose id is 667, has submitted a projection statement:

```
SELECT price, description FROM Product.
```

This projection statement will be rewritten to a form that selects physical fields from the shared table, *Data*. Firstly, the statement will be represented by the following algebraic form:

$$\bar{\pi}_{<price,description>}[667](Product),$$

where $\bar{\pi}$ denotes the projection operation that selects subscripted fields, e.g., *price, description*, from a logical table specified by its name and a tenant id annotation, e.g., *Product* and 667.

Secondly, we look up the $objectId$ from the *Objects* table via the logical table name, (*Product*), and the tenant id, 667. This is specified by using the object name transformation function $\xi^{object}(Product, 667)$, which is assumed to return 1 in this example. With the physical object id and tenant id, we can select all the rows of the tenant's *Product* data from the shared table *Data*, which is specified by the following equation.

$$\sigma_{objectId=1 \wedge tenantId=667}(Data). \quad (1)$$

Thirdly, given the $objectId$, we may obtain the mapping between logical field names and physical field names using the *Fields* table. We specify the mapping via the field name transformation function $\xi^{field}(Product, n_f, 667)$, where the logical field names *id*, *name*, *price*, and *description* are obtained by substituting n_f by *value1*, *value2*, *value3*, and *value4*, respectively. As a result, the logical table *Product* can be reconstructed by appending a rename operation, ρ, and a projection operation in front of the expression in (1):

$$\begin{aligned}[667](Product) = &\rho_{(id,name,price,description)} \\ &\pi_{<value1,value2,value3,value4>} \quad (2) \\ &\sigma_{objectId=1 \wedge tenantId=667}(Data).\end{aligned}$$

Note that the projection operation $\pi_{<value1,value2,value3,value4>}$ is required since the *Data* table has additional fields to keep track of metadata of a record such as *GUID*, *objectId* and *tenantId* fields of the *Data* table, as depicted in Fig.5.

Now that we have reconstructed the logical table *Product* from *Data*, we can apply arbitrary query operations to it:

$$\begin{aligned}\pi_{<price,description>}[667](Product) = &\\ &\pi_{<price,description>} \\ &\rho_{(id,name,price,description)} \quad (3) \\ &\pi_{<value1,value2,value3,value4>} \\ &\sigma_{objectId=1 \wedge tenantId=667}(Data).\end{aligned}$$

Then, the physical form of the tenant-aware logical projection statement can be derived as follows:

```
SELECT price, description FROM (
   SELECT  value1 AS id, value2 AS name,
           value3 AS price, value4 AS description
   FROM Data
   WHERE objectId=1 AND tenantId=667 ).
```

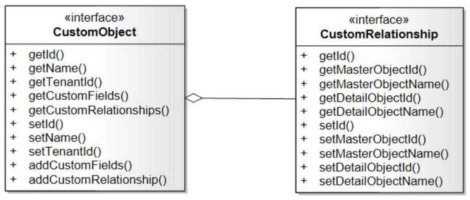


Figure 6: The structure of CustomObject and CustomRelationShip.

Finally, it is important to point out that the rewriting rules are realized in *MultiTenantDataServiceFacade*, as shown in Fig.7, which is the facade for the adapters to the ORM framework. This design made the implementation of rewriting rules easier to be integrated with the ORM framework, which will be explained in detail in the next section.

3.3 Multi-tenant Object-relational Mapping

Because of the difficulties arising from object-relational impedance mismatch (Ambler, 2003), contemporary enterprise applications typically access database through an Object-Relational Mapping (ORM) framework. However, the SQL rewriting mechanisms introduced in the previous section do not deal with the interoperability issues with ORM. Hence, in this section, we propose an transparent approach for integrating SQL rewriting mechanisms into an ORM framework.

Generally speaking, the first step of ORM design is to define the mappings between object fields and database fields. Such mappings are usually specified by the annotations in the source code. To be consistent with the annotation-based convention, we use the annotation *@MultiTenantCapable* to indicate that the annotated object is going to be mapped to a multi-tenant database. For instance, to annotate the *Product* to be multi-tenant capable, the only additional effort is to add an *@MultiTenantCapable* annotation, as shown below:

```
@MultiTenantCapable
Public class Product {...}.
```

Except the annotation, no additional modification is required from the developer's point of view.

We are now ready to introduce the underlying techniques of the proposed approach. In order to map user-customized domain objects into Universal Table schema layout, metadata information such as class name, field name and relationship has to be extracted and then attached to the mapping object. As depicted in Fig.6, we defined two general interface, namely, *CustomObject*, *CustomField* and *CustomRe-*

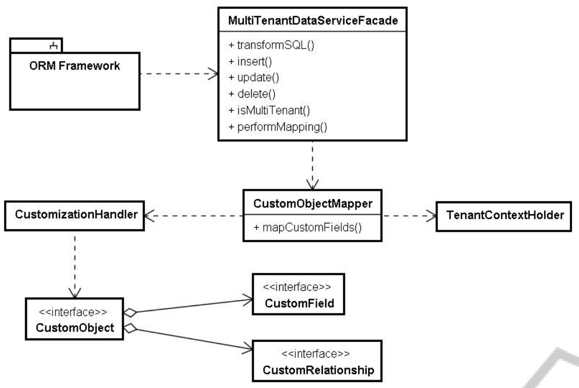


Figure 7: The overall design of multi-tenant ORM.

lationship, to store the metadata information mentioned above. At runtime, the system periodically checks any newly added user-customized domain objects. If the annotation @*MultiTenantCapable* is observed, then the annotated object will be enhanced to implement the *CustomObject* and related interfaces on the fly by a bytecode rewriting mechanism to be sketched below. In particular, the implementation of the mappings between user-customized domain objects and the underlying schema layout is dynamically generated and injected into the bytecode of these objects.

Figure 7 displays the overall structure of how the proposed mechanism adapts to an existing ORM framework. There is a class called *MultiTenantServiceFacade* that serves as a unified entry point so that the proposed mechanism is more portable to different ORM frameworks. The SQL rewriting rules presented in the previous section in are implemented in the insert, update, delete methods of *MultiTenantServiceFacade*. The object-relational mapping tasks are delegated to *CustomObjectMapper*, which uses *CustomizationHandler* to interact with user-customized domain objects. It is worth mentioning that, due to the use of thread specific storage pattern, *CustomObjectMapper* and *CustomizationHandler* are able to obtain the reference to tenant contexts in situ without any parameter passing.

Currently, we implement the proposed design based on JavaAgent (Aarniala, 2005), as the bytecode transformation tool and DataNucleus's JDO (Russell, 2010) implementation, as the underlying ORM framework. DataNucleus (Miller et al., 2010) is designed based on OSGi platform (Hall et al., 2011) so that our extension can be easily integrated into it as a bundle. The main transforming logic is implemented in a specific class called *MTAClassFileTransformer* which is initialized by JavaAgent and is hooked in JVM. Before the annotated classes are loaded, JVM will delegate to *MTAClassFileTransformer* so that it has a chance to modify the bytecode.

4 EVALUATION

This section presents the preliminary results of the feasibility study and performance evaluation of our approach.

4.1 Feasibility

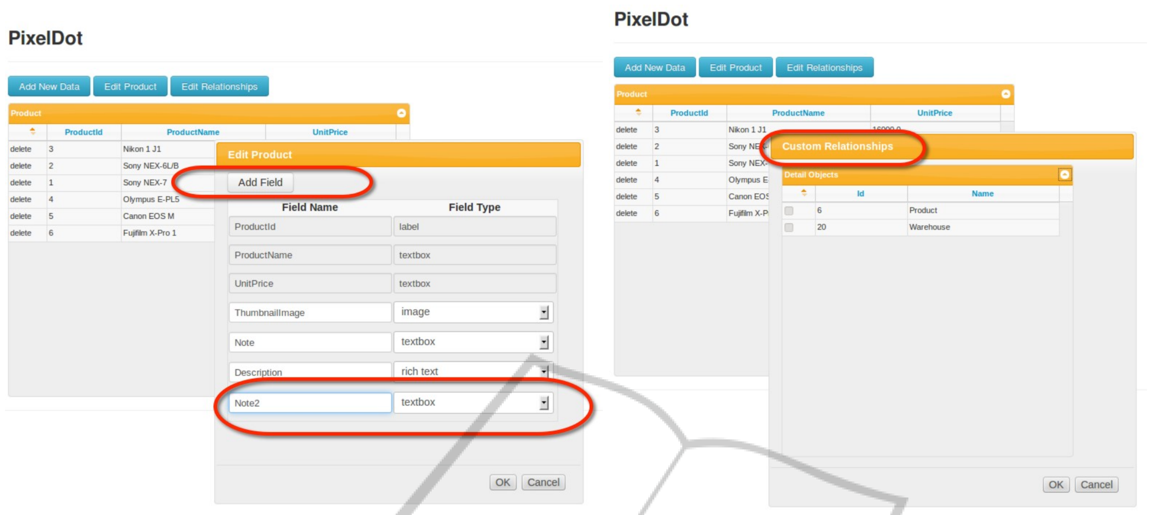
We studied the feasibility of the proposed service framework by developing a Java-based prototype. To verify the prototype, we also implemented a simple SaaS application called ShoppingForce.com on top of the service framework. The SaaS application is able to access the underlying tenant context management, ORM, and schame-mapping services.

In the application layer, we realize the thread-specific storage via a static member variable, which is realized by the *ThreadLocal* class provided by JDK. Tenant contexts belonging to different threads are isolated by *ThreadLocal*. In other words, although thread-specific storage seems to be global to the system, when a thread accesses it by calling the *getContext* method, only the tenant context that is specific to the calling thread is returned. In the data layer, to access the physical schema, the application uses JDOQL (JDO Database Query Language) (Russell, 2010) and manipulates JDO API. Then, The JDOQL is translated internally to SQL statements and then used as the inputs of the proposed rewriting schemes.

To create a virtual application, the tenant applies for an account on-line and a tenant profile is then generated accordingly. We have created two different on-line shopping applications hosted on ShoppingForce.com. Sometimes, a tenant needs to modify default virtual schema such as adding tenant-specific columns. In such case, ShoppingForce.com provides a set of schema customization pages which can be accessed from the account management page, as shown in Fig. 8.

4.2 Performance

To test the performance of the tenant-aware schema layout management service in the data layer, we conducted experiments in a stand-alone switched network. In the network, the test client and the test server are deployed on two separate PCs with Intel Core i7 3.4-GHz processor with 4G bytes memory. For the test client, we use Apache JMeter 2.9 (Halili, 2008), a



	ProductId	ProductName	UnitPrice
delete	3	Nikon 1 J1	
delete	2	Sony NEX-6L/B	
delete	1	Sony NEX-7	
delete	4	Olympus E-PL5	
delete	5	Canon EOS M	
delete	6	Fujifilm X-Pro 1	

Field Name	Field Type
ProductId	label
ProductName	textbox
UnitPrice	textbox
ThumbnailImage	image
Note	textbox
Description	rich text
Note2	textbox

	Id	Name
	6	Product
	20	Warehouse

Figure 8: The object customization and relationship customization pages in ShoppingForce.com.

well-known open source and general-purpose performance measurement platform, which can be used to simulate arbitrary load types on the server or network to test overall performance under different load types. The service framework and the database is deployed on the test server, where the database is MySQL Community Server 5.7 with InnoDB engine running on Ubuntu Linux 12.04.

For the experiments, we set up a imaginary scenario, in which there are 100 tenants and each tenant stores 100,000 records in both a Universal Table implementation and a Private Table implementation. In the Universal Table implementation, there are respectively 10 million, 10 million, and 5 million records stored in the *Indexes*, *UniqueFields*, and *Relationships* table. For each tests, based on the above scenario settings, several concurrent threads that issue query requests to the service framework. After a request is finished, the JMeter platform gathers the responded results and reports the turnaround time. We performed experiments for Selection, Projection, and Join statements of Private Table and Universal Table implementations, respectively. When performing the tests, built-in cache mechanism of query processer is turned off to better reflect the actual overheads induced by the transformed SQL.

A summary of experimental results for Private Table and Universal Table is shown in Table 1. Compared to the Private Table implementation, which serves as a baseline, there is a great performance penalty for ”multi-tenant-ifying” the database. The main reason is that schema-mapping involves overheads of additional database access since all meta information of logical-physical mapping has to be stored in physical storage. However, we believe that

Table 1: Average turnaround time of queries on Private Table and Universal Table schema layouts. (in milliseconds).

Operation Type	Private Table (Baseline)	Universal Table (SQL)	Universal Table (ORM)
Select	0.4469	5.1793	14.0067
Insert	1.3823	13.7154	14.4246
Delete	0.5264	10.2565	16.0911
Update	0.5238	9.1029	15.3657

the overhead is acceptable for most enterprise SaaS applications because the worst turnaround time of query operations is still less than 20ms. Moreover, the performance can be improved significantly if the built-in cache mechanism of query processor is turned on. It is also worthy to point out that the turnaround time for ORM implementation is a bit slower than the direct SQL implementation. This result reflects the trade-offs between code maintainability (via the use of ORM framework) and performance.

5 CONCLUSION

In this paper, we have investigated the design of a multi-tenant service framework for developing enterprise SaaS applications. The service framework addresses three essential design aspects, namely, tenant context storage and propagation, schema-mapping, and the integration of ORM framework, of enterprise SaaS applications. We have also presented a prototype implementation of the proposed approach and conducted performance evaluations to assess the overheads. In addition, a sample multi-tenant SaaS application, the ShoppingForce.com and two tenant-specific virtual applications are also constructed to demonstrate the feasibility of the service framework.

Moving ahead, further research is clearly required to investigate approaches for enhancing the security aspect of the proposed framework. Essentially, multi-tenancy promotes resource sharing, which unavoidably trades a certain amount of security for the lower service costs. Fortunately, security issues caused by resource sharing can be significantly reduced if the multi-tenant SaaS application is deployed on a middleware platform that employs advanced access control and program monitoring mechanisms for intercepting unauthorized accesses to a shared resource. Hence we shall look into those mechanisms and investigate how to leverage them to prevent unauthorized data accesses, such as checking tenant ID's. On a different front, we are going to explore more transparent ways, such as aspect-oriented programming or dependency injection, to help developers transform a single tenant enterprise application into a multi-tenant one based on the proposed service framework with less efforts.

ACKNOWLEDGEMENTS

This work is partially sponsored by Ministry of Science and Technology, Taiwan, under grant 102-2221-E-035-039 and 102-2627-E-002-001.

REFERENCES

Aarniala, J. (2005). Instrumenting java bytecode. In *Seminar work for the Compilerscourse, Department of Computer Science, University of Helsinki, Finland*.

Ambler, S. (2003). *Agile database techniques: Effective strategies for the agile software developer*. John Wiley & Sons.

Aulbach, S., Grust, T., Jacobs, D., Kemper, A., and Rittinger, J. (2008). Multi-tenant databases for software as a service: Schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*.

Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., and Fremantle, P. (2010). Multi-tenant soa middleware for cloud computing. In *Cloud computing (cloud), 2010 ieee 3rd international conference on*. IEEE.

Bezemer, C.-P. and Zaidman, A. (2010). Challenges of reengineering into multi-tenant saas applications. *Delft University of Technology, Tech. Rep. TUD-SERG-2010-012*.

Cai, H., Wang, N., and Zhou, M. J. (2010). A transparent approach of enabling saas multi-tenancy in the cloud. In *Proceedings of IEEE World Congress on Services*.

Chong, F. and Carroro, G. (2011). Architecture strategies for catching the long tail. In *Available at: http://msdn.microsoft.com/en-us/library/aa479069.aspx*.

Fang, S. and Tong, Q. (2011). A comparison of multi-tenant data storage solutions for software-as-a-service. In *Proceedings of the 6th International Conference on Computer Science and Education(ICCSE 2011)*.

Galchev, G., Fleischer, C., Luik, O., Kilian, F., and Stanev, G. (2007). Session handling based on shared session information. US Patent App. 11/322,596.

Halili, E. H. (2008). *Apache Jmeter: a practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing.

Hall, R., Pauls, K., and McCulloch, S. (2011). *OSGi in Action: Creating Modular Applications in Java*. Manning Publications Company.

Joines, S., Willenborg, R., and Hygn, K. (2003). *Performance Analysis for Java Web Sites*. Addison Wesley.

Koziolek, H. (2012). The sposad architectural style for multi-tenant software applications. In *Proceedings of the 9th Working IEEE/IFIP Conferences on Software Architecture*.

Krebs, R., Momm, C., and Konev, S. (2012). Architectural concerns in multi-tenant saas applications. In *Proceedings of the International Conference on Cloud Computing and Service Science (CLOSER12)*.

Li, C. (2010). Transforming relational database into hbase: A case study. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*. IEEE.

Liao, C.-F., Chen, K., and Chen, J.-J. (2012). Toward a tenant-aware query rewriting engine for universal table schema-mapping. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*.

Miller, F., Vandome, A., and John, M. (2010). *DataNucleus*. VDM Publishing.

Momm, C. and Krebs, R. (2011). A qualitative discussion of different approaches for implementing multi-tenant saas offerings. In *Proceedings of Software Engineering 2011, Workshop*.

Pereira, J. and Chiueh, T. C. (2007). *SQL Rewriting Engine and its Applications, Technical Report*. Stony Brook University.

Russell, C. (2010). *Java Data Objects 2.0. JSR 243 Specification.*

Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (1996). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons.

Shimamura, H., Soejima, K., Kuroda, T., and Nishimura, S. (2010). Realization of the high-density saas infrastructure with a fine-grained multitenant framework. *NEC Technical Journal*, 5(2).

Strauch, S., Andrikopoulos, V., Sáez, S. G., Leymann, F., and Muhler, D. (2012). Enabling tenant-aware administration and management for jbi environments. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*. IEEE.

Truyen, E., Cardozo, N., Walraven, S., Vallejos, J., Bainomugisha, E., Gunther, S., D'Hondt, T., and Joosen, W. (2012). Context-oriented programming for customizable saas applications. In *Proceedings of ACM Symposium on Applied Computing*.

Wang, H. and Zheng, Z. (2010). Software architecture driven configurability of multi-tenant saas application. In *Proceedings of International Conference on Web Information Systems and Mining*.

Weissman, C. D. and Bobrowski, S. (2009). The design of the force.com multitenant internet application development platform. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*.