Enabling Legacy Applications for Multi-tenancy Without Reengineering

Uwe Hohenstein(✉) and Preeti Koka

Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81730 Munich, Germany
{Uwe.Hohenstein,Preeti.K}@siemens.com

Abstract. Multi-tenancy is an architectural style to share resources amongst several tenants. It is an important facet of Cloud Computing and often considered a key element to make Software-as-a-Service (SaaS) profitable. Indeed, SaaS providers adopt multi-tenancy to optimize resource usage and to save operational costs. While literature often discusses how to develop new, green-field software with multi-tenancy, this paper focuses on adding multi-tenancy to existing, brown-field software. This is particularly relevant in the context of Cloud migration where legacy software should be moved into the Cloud. The major contribution of this paper is to present an approach to leave the application's source code untouched, i.e., to add some new components in order to enable the application for multi-tenancy. To this end, we apply the aspect-oriented language AspectJ in an industrial case study to evaluate what can be achieved with such an approach as well as to enumerate the benefits and drawbacks in detail. In a nutshell, the approach is appropriate to handle REST applications and/or backend services. The following important facets of multi-tenancy can be achieved: Tenant management; tenant-specific authentication and data isolation among multiple tenants for various database servers and strategies; tenant-specific customization by modifying existing behavior, particularly, removing functionality but also to introduce new functionality; and as a by-product, to monitor all tenants' activities as a prerequisite for a tenant-specific billing.

Keywords: Multi-tenancy · Cloud migration · Aspect-orientation · AspectJ · Industrial application · Case study

1 Introduction

The NIST definition [21] defines Cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". Software-as-a-Service (SaaS) is thereby one service model besides Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). SaaS is a delivery model that enables customers, the so-called *tenants*, to lease and use services without buying a software license and setting up a local installation [17]. Moreover, tenants pay only for what they use to what extent according to the pay-as-you-go principle.

E. Cabello et al. (Eds.): ICSOFT 2016, CCIS 743, pp. 284–308, 2017.
DOI: 10.1007/978-3-319-62569-0_14

The goal of SaaS providers is to save operational cost in order to be competitive by means of an efficient utilization of hardware and software resources and improved ease of maintenance [3]. This let SaaS providers usually adopt a multi-tenant architecture [5]. Multi-tenancy is a software architecture principle that lets several tenants share a common infrastructure. It is widely agreed that a well-economical SaaS application has to pursue a multi-tenant architecture.

Since software is more and more becoming an on-demand service drawn from the Cloud, industries are interested in offering SaaS to enter new businesses. Having a huge amount of legacy applications, there is a strong interest in moving these applications into the cloud first for entering SaaS business while preserving investments. As a side effect, applications can also benefit from features such as elasticity and pay-as-you-go. But industries have the challenge to convert legacy applications into multi-tenant SaaS without spending too much time and effort on refactoring [4].

Several papers such as [3, 29] discuss multi-tenant architectures with pros and cons according to what is shared by the tenants: the topmost web frontend, middle tier application servers, the underlying database. Others, e.g., [1], define further degrees of sharing and categorize migration types to cloud-enable applications.

Striving for multi-tenancy, the SaaS provider has to balance between easy implementation and saving operational costs by efficient resource utilization. The simplest approach to make an application multi-tenant with lowest development effort is a virtualization approach [14]. This approach let each tenant obtain a virtual machine (VM) containing an application server, the application and a DB server. The ease of this approach is paid by a higher consumption of resources and higher costs especially in public clouds where each VM (one for each tenant) has to be paid. Even on premises more equipment than necessary has to be provided. Moreover, each tenant requires a database server license or additional costs for using a database as a Cloud service.

At the other edge of the scale, fully efficient multi-tenancy [5] let all the tenants share all resources: one Tomcat, one application, and one database server amongst all tenants. However, a significant re-engineering of applications is required to set up a fully multi-tenant application, thus leading to high development costs [23].

Recent technologies facilitate further approaches. Particularly, container technologies enable another approach to make application deployments multi-tenant, lying in between the previous two extremes. This implementation involves deploying the application stack (application server, application and database server) on a separate container for each tenant. Containers such as Docker are more light-weight than virtual machines, but still produce more load and thus require more resources than fully efficient multi-tenancy. As an advantage, tenant-specific customizations become easier since each container can be equipped with a different software variant.

In fact, SaaS applications have to be customizable or configurable to fulfil the varying functional requirements of individual tenants [14]; customers want to add or modify specific features. From the SaaS provider's view, various degrees of feature sets could be offered with different prices, especially a “freemium” version with a reduced function set. Several papers such as [2, 9] recognize tenant customization as one important requirement and challenge, and [17] states that it is not trivial to adapt the business logic and data to the requirements of the different tenants. Most work on customization focuses on product-line approaches [26] to offer variability. Using aspect-oriented programming (AOP) is sometimes proposed to achieve configurability, e.g., by [29, 34].

In this paper, we also apply AOP, however, at a broader scope to migrate existing applications into fully multi-tenant SaaS applications. We investigate how to benefit from the aspect-oriented language AspectJ [16] in this context. We have a clear idea in mind: To add multi-tenancy to existing applications without any reengineering and without explicitly modifying the source code. To explore our idea, we use an existing industrial application that was originally not developed for a multi-tenant environment and serves users of exactly one tenant. More precise, each tenant obtains one dedicated application instance deployed on a Tomcat application server and using an Oracle database on premise so far. That is, the application is managed per tenant similar to traditional application service providers. We then elaborate upon the feasibility to realize the AOP idea thereby illustrating the major advantages of our migration approach. Hence, this paper takes a practical view on Cloud migration and presents a low-effort approach for offering legacy applications as multi-tenant SaaS in a Cloud.

In a nutshell, it is possible to achieve tenant isolation, to modify existing behavior in a tenant-specific manner, to introduce new services for specific tenants, and to monitor requests per tenant for billing purposes. Enabling such multi-tenancy facets is achieved without explicitly touching source code or building a new application; only a restart of Tomcat is required after having deployed some additional components. Hence, we obtain a simple and cheap mechanism by only adding components to existing applications – without any further reengineering and refactoring of source code.

The motivation for our work is manifold. The approach is a first step to let existing applications become Cloud-ready and to enable entering the SaaS business fast and easily. Such a first trial can explore SaaS business opportunities, maybe offering reduced functionality, and to expand business to a larger customer base with low expenses. Being easily applicable to other applications, our solution reduces time-to-market and saves development effort. And finally, free demo versions of existing applications can be made publicly available in a Cloud as a teaser. Since no profit can be directly made in that case, we benefit from small investments in development.

This paper is an extended version of [10] where we have already outlined the basic ideas for the previously mentioned software running in Tomcat and using an Oracle database. Here, we extend the mechanism to cover other database servers and further data isolation strategies according to [6].

The remainder of this paper is structured as follows. Section 2 presents related research and deduces the necessity for this work. Before discussing the migration approach in depth, we give in Sect. 3 a short introduction into the aspect-oriented AspectJ language, as far as it is necessary to understand how we applied AspectJ. Section 4 introduces the application, which we used in a concrete industrial study to prove effectiveness, in its original single-tenant form. We present our approach to migrate to a multi-tenant Cloud application with low programming effort in detail by discussing the components that implement important facets of multi-tenancy such as tenant isolation and customization. In Sect. 5, we elaborate upon the flexibility of the AspectJ approach to cover other database servers and further data isolation strategies in addition to our previous work [10]. Section 6 presents an evaluation of the AspectJ approach with regard to implementation effort, modularity, and adaptability. Moreover, the lessons learned are discussed. Finally, the conclusions summarize the discussion and presents future ideas.

2 Related Work

A lot of recent research focuses on migrating legacy applications into the cloud, e.g., suggesting checklists and methodologies to perform migrations. For example, ARTIST [24] provides methods, techniques, and tools to guide companies in moving applications into the cloud in three phases pre-migration, migration, and post-migration. The approach attempts to better support the complex, time-consuming, and expensive tasks during migration.

Decisions to migrate existing services to the cloud can be complicated as the benefits, risks, and costs of using the Cloud are complex. [11] states that a migration should also consider organizational and socio-technical factors. Their Cloud Adoption Toolkit offers a collection of tools for decision support and helps to identify relevant concerns and match them to appropriate technologies. A particular cost modeling tool can be used to compare the cost of different cloud providers and deployment options. A case study presents in detail this tool.

Binz et al. [4] discuss vendor lock-in as a major difficulty for migrating existing applications into and between different clouds. The CMotion framework models entities and their dependencies as a basis for supporting migration. Anyway, adapters have to be implemented manually.

This work in the area of cloud migration is quite general and does not address the integration of multi-tenancy into legacy applications. Indeed, our approach combines migrating applications to the Cloud with adding multi-tenancy to legacy applications.

In fact, the specific topic of multi-tenancy is often considered as a challenge in research. Several papers, for example [6] and [15], describe the possible variants of multi-tenancy. Momm and Krebs [23] consider approaches to reduce resource consumption and discuss some cost aspects of sharing. Wang et al. [33] make recommendations on the best multi-tenant variant to use based on the number of tenants, the number of users per tenant, and the amount of data per tenant. Guo et al. [9] discuss the implementation principles for application-level multi-tenancy and explore different approaches to improve isolation of security, performance, availability, and administration. Fehling et al. [8] come up with prospects for the optimization of multi-tenancy by distributing the tenants with respect to Quality of Service.

Bezemer et al. [2] present an architectural approach for reengineering applications to enable multi-tenancy in software services. The discussion especially considers a multi-tenancy reengineering of workflow and UI configuration. A specific multi-tenancy reengineering pattern takes into account a multi-tenant database, tenant-specific authentication, and configuration. This reengineering pattern is applied to an existing single-tenant application in a case study. However, the reengineering effort was relatively little due to a well-designed and layered architecture. In an additional work [3], they manually transform the ScrewTurn wiki case to a multi-tenant application and encounter security, data protection, data isolation, configurability, performance isolation of tenants, and scalability issues for tenants from different continents, as the core challenges. Unfortunately, the authors do not solve all the previously mentioned issues. Beside implementation effort, they consider the recurrence of maintenance tasks such as patches or software updates as another driver for operational cost.

In contrast to this general work on strategies and their impact on resource consumption, our approach tackles the problem of adding multi-tenancy by avoiding reengineering efforts.

Further research considers tenant-specific customizations as an important requirement for multi-tenancy, e.g., case studies such as [15, 18] which try to configure multi-tenant applications for tenants. The elements of an application that need to be customized are graphical user interface, business logic, service selection and configuration, and data [30]. Customization could be performed in two ways [29]: A source-code based approach allows customizing SaaS applications by integrating new tenant-specific source code. Such an approach has been pursued by [13, 35]. In spite of giving tenants more flexibility in the customization process, this approach has several disadvantages. At first, each tenant must know the implementation details of the SaaS application. Then, security regulations of the application might be violated if tenants are able to integrate source code. Since all the tenant-specific extensions have to be retained, software upgrades become more complicated for the SaaS provider. [32] considers source code based approaches as too complex.

An alternative composition-based approach let SaaS applications be customized by composing variants. An application template contains customization points [20], i.e., unspecified parts, which can be configured by selecting predefined components from a provided set [19, 22, 25].

Adopting work from the area of product-line engineering, Pohl et al. [26] point out four key concerns to be addressed for customization: modeling customization points and variations, describing relationships among variations, validating customzations performed by tenants, and dis-/associating variations from/to customization points during runtime.

Shahin et al. [29] tackle all these concerns and propose the Orthogonal Variability Modeling (OVM) to model customization points and variations and to describe the relationships among variations. Tenants' customizations are validated by a Metagraph-based algorithm. An aspect-oriented extension of the Business Process Execution Language (BPEL) is used to associate and disassociate variations to/from customization points at run-time. The approach is illustrated by a Travel Agency example.

Three of the above concerns are dealt with by [20]. They also use Metagraphs to model customization points, variants, and their relationships. Moreover, they propose an algorithm to validate customizations made by tenants. [30, 31] handle only the modeling of customization points and variants using an ontology-based customization framework with OVM. Tenants are guided through the customization process to avoid unpredictable customizations.

Walraven et al. [32] investigate middleware component models with respect to offering software variations to different tenants and come to the conclusion that support is too inflexible. Using Google AppEngine, they propose a multi-tenancy support layer that combines dependency injection with middleware support. They evaluate operational expenses and flexibility for an online booking scenario. The approach requires that dedicated customization points are inserted into the code for applying customization. Similarly, Wang and Zheng [34] apply aspect-orientation in a case study, but still rely on preparing the software architecture accordingly.

In spite of providing interesting insights in multi-tenancy and configurability, all this research starts from green-field or need to insert customization points in the existing application. In contrast, our approach leaves the original application unchanged. To our knowledge, there is also no work combining an approach to migrate applications to the Cloud with adding multi-tenancy for a legacy application by avoiding major code changes.

3 Aspect-Oriented Programming in AspectJ

Aspect-orientation (AO) is a paradigm that helps to develop software in a modular manner [12]. AO provides systematic means for effectively modularizing crosscutting concerns (CCCs). CCCs are those functionalities that are typically spread across several places in the source code and often lead to lower programming productivity, lower degree of code reuse, and poor traceability and quality [7]. Special aspect-oriented languages offer advanced concepts to modularize CCCs and to avoid the well-known symptoms of non-modularization such as code tangling and code scattering.

Our approach relies on the AspectJ language [16]. AspectJ is an extension of the Java language introducing a new concept of aspect to Java. An *aspect* changes the dynamic structure of a program by intercepting certain points of the program flow, the so-called *join points*. Join points can be method and constructor calls or executions, field accesses, and exceptions etc. *Pointcuts* syntactically specify those join points in the flow by means of a signature expression. The actions to be taken before and/or after the join points are defined by *advices*.

AspectJ, as presented in [16], is a language of its own, in fact, an extension of Java. Hence, it requires a dedicated AspectJ compiler. Usually, the AJDT plug-in will be installed in Eclipse. However, an AspectJ compiler requires changes in the build process, which is often not desired, so for us: We do not want to re-compile the existing application. Then, using Java annotations is an alternative. The following is an example for a simple aspect with annotations:

```
@Aspect class MyAspect {
  @Before("execution(* MyClass*.get*(..))")
  public void myAdvice() {
    do something in Java before specified join points
} }
```

An annotation `@Aspect` lets the Java class `MyAspect` become an aspect. The method `myAdvice` is a `@Before` advice that adds Java logic before those joinpoints that are captured by the pointcut. The pointcut is specified within `@Before` as a string: Any execution of any method starting with `get`, having any parameters and any return type, belonging to a class starting with `MyClass`. Wildcards can be used to determine several methods of several classes. A star “*” in names denotes any character sequence; “*” used as a type stands for any type. Parameter types can be fixed with data types or left open `(..)`. Similarly, `@After` and `@Around` advices can be used to execute an advice after or around join points, resp. An aspect can also declare attributes and methods; it can also extend another aspect.

This is only a very brief overview of AspectJ. Concrete examples will be discussed in the successive section. It is important to note that this is pure Java code that runs with any Java compiler. So-called load-time weaving (LTW) lets the advices be woven into the code whenever a class is loaded by the class loader.

4 Adding Multi-tenancy to Existing Applications

In this paper, we use an existing industrial Java application that provides customers a REST service in the travel management domain. The application runs in a Tomcat application server and uses an Oracle database (DB) for storing data at the backend. Currently, the application is shipped as a single-tenant application to individual customers and deployed at the customer site. Thus, each customer obtains a full application stack consisting of Tomcat, the application, and an Oracle database server.

The intention is to deploy this application in a public cloud thereby enabling it for multi-tenancy. This means that Tomcat, the application and the database have to be shared amongst several tenants. Further details about the application are subject to confidentiality and irrelevant for the message of this paper.

4.1 Tomcat and Oracle Basics

Tomcat and Oracle have some specific concepts the understanding of which is necessary for the remainder of this paper. Tomcat provides several forms of user authentication, a form-based for Web application, basic authentication for REST services etc. Having enabled authentication, Tomcat shields the application by asking for a user and a password. User, passwords, and user roles are stored for each application in a configurable "user/roles store" like an XML file, a relational DB, a JDNI store etc. When a user logs in to an application, the Tomcat container checks that store for valid credentials. The application can also restrict functionality to users with specific roles. The legacy application we use in our case study applies Tomcat's basic authentication. To this end, a dedicated schema `Auth(entication)` in Oracle contains the user/roles tables. The connect string with a specific user/password is part of the Tomcat configuration file.

In Oracle, each user requires a password to login; the user obtains an associated DB *schema* with the same name. Every user can create the same set of tables with the same statement – in his schema. Thus, an Oracle user/schema corresponds to a tenant "database". Schemas are isolated from each other. To access data in another schema (i.e., of another user), tables can be prefixed by a schema name. However, the owner of the foreign schema must explicitly grant access to the user. In the following, we use the notion `schema.table` to refer to a table in a specific schema.

A *database instance* is the Oracle notion of a database server. Such an instance has exactly one *database* being associated. A JDBC driver connects to that database.

4.2 Tenant and User Management

The major concern of this paper is to enable an existing Tomcat application for multi-tenancy, i.e., to share the Tomcat application server, the application, and the Oracle database instance amongst tenants. According to [2], one important prerequisite for multi-tenancy is an appropriate tenant/user management. In particular, the following workflow should be supported:

1. Tenants must be given a possibility to register for using the application.
2. A SaaS administrator should be able to approve or deny the tenant for using the application depending on whether a contract about payment details has been set up between the SaaS provider and the tenant.
3. If the SaaS provider has approved a tenant, the tenant obtains a dedicated database. Moreover, the tenant is allowed to register its users.
4. All the registered tenants' users should be able to use the application.

4.3 Initial DB Setup for Multi-tenancy

The original application keeps its data in a database schema. Indeed, there might be several, however, we collapse them to one referred to as `Appl`. We assume another schema, referred to as `Auth`, which contains the Tomcat authentication tables `Users` and `User_Roles` with Tomcat users with their roles. Tomcat accesses these tables to check the password for any login to the application during authentication. Only Tomcat users in the `Users` table can authenticate.

During an initial database setup, the `Users` table in the `Auth` schema is extended with a column `tenant` to keep the association between a user and the tenant s/he belongs to. Moreover, a new Oracle user/schema `Admin` is created that is exclusively used by the SaaS administrator to keep information about tenants. The setup also creates a new table `Tenants` in this schema to keep registered tenants with their administrators and a `UserMonitoring` table for monitoring purposes (cf. Sect. 4.7).

The newly introduced tenant administration service (cf. Sect. 4.4) requires a SaaS administrator to perform tenant management. To this end, a new Tomcat user `SaaS` with a new role `SaaS` is added to the `Users` and `User_Roles` tables.

Finally, an SQL script `createApplicationTables.sql` is required to create all the application's tables in any new tenant schema.

All these steps do not affect the existing application, but only require some SQL scripts to be executed.

4.4 Tenant Administration Service

New services are required for tenant administration purposes, especially for registering tenants and users, to support the workflow in Subsect. 4.2. The existing application source code is not affected. Hence, we implemented a new REST server to provide corresponding functionality:

1. `POST TenantService` allows a tenant to register for using the application. Everybody is allowed to invoke this service. The request payload has to specify a name `Tenant`*X* for the tenant and an administrator by name and password. Name and password are required to let a tenant register users in Step 3. This information is stored in a table `Admin.Tenants(name, admin, password, approved, …)`.
2. `PUT TenantService/Tenants/{Tenant`*X*`}` can be used by the SaaS administrator to enable or disable access for `Tenant`*X*; the request body contains `{"approve":Yes}` or `{"approve":No}` accordingly. Only the SaaS administrator is allowed to invoke the service. To this end, we set up a new organizational Tomcat role `SaaS` which allows the administrator to manage tenants. If the SaaS administrator approves TenantA, then `approved=1` is set for TenantA in the `Admin.Tenants` table; the record for TenantA's admin is copied from `Admin.Tenants` (cf. Step 1) to the `Auth.Users` table. Moreover, the tenant administrator obtains a new Tomcat role `TAdmin` in the `Auth.User_Roles` table. This role allows him to register tenant's users for the application. The approved tenant obtains an Oracle user and schema TenantA, i.e., a database to keep the tenant's application data isolated. Finally, all the application tables are created in the new schema by executing the SQL script `createApplicationTables.sql` in schema TenantA.
3. `POST TenantService/Tenants/{Tenant`*X*`}` lets the tenant administrator for `Tenant`*X* create a user to make a user known to the application. The invoker requires the `TAdmin` Tomcat role for Tomcat authentication. The request payload specifies the name of the user and a password, which both are inserted to the `Auth.Users` table. Furthermore, the user obtains one or more roles, which enables him to use the application with the above credentials. The association of a user to his tenant is stored in the `tenant` column of the `Auth.Users` table. Table 1 shows the contents of the `Users` and `User_Roles` tables after the administrator AdminA for TenantA has registered UserA1 and UserA2; an explanation describes when each record has been added.
4. The users UserA1 and UserA2 of TenantA are then able to login to the application and to use it.

Table 1. Database contents for authentication (adopted from [10]).

Users	**user_name**	**user_pass**	**tenant**	
	… existing users	…	NULL	
	SaaS	SaaS	NULL	in 4.3
	AdminA	PwA	TenantA	Step 2
	UserA1	PwA1	TenantA	Step 3
	UserA2	PwA2	TenantA	Step 3

User_Roles	**user_name**	**role_name**	
	… existing users	… existing roles	
	SaaS	SaaS	in 4.3
	AdminA	TAdmin	Step 2
	UserA1	User	Step 3
	UserA2	User	Step 3

These services can simply be deployed as a new application in Tomcat in order to become immediately effective. Services rely on the following Tomcat roles giving privileges to the various types of users:

- A new `SaaS` role for the administrator of the SaaS applications to perform administrative tasks such as tenant approval;
- a new `TAdmin` role for a tenant administrator to enable registering tenant's users;
- `User` for the users of the application: Indeed, there may be several with specific privileges. For the ease of discussion, we collapse them to one `User` schema.

4.5 Data Isolation

Tenants and their users are now known to Tomcat and allowed to access the application since Tomcat authenticates against the `Auth.User/UserRoles` tables. However, *all* these users access the same application and use the original tables in the `Appl` schema. Hence, there is no effective data isolation between different tenants as requested by [9]. To achieve data isolation, a user's data must be stored in the tenant's schema (i.e., database). This means that every database access of a logged-in user must be re-directed to the correct tenant schema. In fact, AspectJ comes here into play since it enables intercepting every user authentication without explicitly modifying, recompiling, or rebuilding the original application. Then, the user can be determined and the corresponding Tenant*X* for the user derived. The following code sketches a corresponding AspectJ aspect:

```
@Aspect public class MTE {
  @Around(
      "execution(* com.siemens.app.ExistingAppl.svc*(..))
       && !within(com.siemens.aspects.MTE)")
  public Object interceptRequests
                  (ProceedingJoinPoint jp) {
     (1) determine user from HTTPRequest and
         derive role & tenant (from Users table);

     (2) store user/tenant/role for later usage;

     (3) switch acces to tenant database;

     return jp.proceed(jp.getArgs()); /* call original
                                         logic of svc* method */
} }
```

We do not use the AspectJ language and its compiler because we do not want to change the build process. Instead, we rely on pure Java with AspectJ annotations and load-time weaving. The annotation `@Aspect` let the Java class `MTE (MultitenancyEnabler)` become an aspect. The method `interceptRequests` is annotated with `@Around` and defines an advice to be executed at join points. These join points are specified by a pointcut string within the `@Around` annotation. The advice intercepts any execution of methods starting with `svc`… belonging to the class

ExistingAppl (i.e., the basic REST service) with any parameters (..) and returning any type (*). We could also specify several method signatures individually and combine them with '||' (logical OR) without wildcards.

The @Around method interceptRequests implements the logic to be executed at each join point, i.e., any execution of a svc... method specified by the pointcut, and replaces the original behavior with its body. The parameter jp of type ProceedingJoinPoint is used to execute the original logic at the join points in the advice by means of jp.proceed(). Furthermore, jp also gives access to the context of invocation such as the parameter values (jp.getArgs()) and the signature of the concrete svc... method (jp.getSignature()). Since the method is implicitly invoked by jp.proceed() inside the aspect, an endless loop will occur. This is avoided by adding !within(MTE) in the pointcut to not intercept any invocation that occurs within the aspect itself.

Please note only the pointcut "execution(* com.siemens.app. ExistingAppl.svc*(..)" of advice interceptRequests depends on the application code. This pointcut specifies what methods or services are intercepted, here of class ExistingAppl that implements the REST service.

One open point now is how to get the user name from Tomcat authentication (cf. (1) in the code above). Unfortunately, there are several ways to pass the authentication context to the application, and it is unknown what mechanism has been used in the original application. For instance, the application can declare a HttpServletRequest req variable. Such a variable declaration can be annotated with @Context in a service class which let the value be injected by the Tomcat container. The HttpServletRequest can then be used to derive authentication information, e.g., by req.getUserPrincipal().getName(). Another way is to specify an additional @Context HttpServletRequest parameter in a service method. Besides not knowing the used mechanism, even a global variable req is usually private and not accessible from an external aspect.

Investigating the behavior of Tomcat, we noticed that Tomcat invokes for authentication in any case a _handleRequest method of a class WebApplicationImpl. Thus, a @Before advice in the MTE aspect can intercept the method execution in order to extract the HttpRequest and then the user name:

```
@Before("execution
        (* com.sun.jersey.server.impl.application
         .WebApplicationImpl._handleRequest(..))
         && this(w) && !within(com.siemens.aspects.MTE)")
public void getUserInfo(JoinPoint jp,
                        WebApplicationImpl w) {
  String user = w.getThreadLocalHttpContext()
              .getRequest().getUserPrincipal().getName();
  determine role and tenant for user;
}
```

Please note AspectJ is able to intercept JARs, even of 3rd party tools like Tomcat without having the source available!

The clause `this(w)` binds the variable `w` to the called object of type `WebApplicationImpl`. The method `getThreadLocalHttpContext()` is used to get the request-local `HttpContext`, which is then used to derive the `HttpRequestContext` and the `Principal` of the user who has logged in. The tenant to whom the user belongs can be determined by using the `Auth.Users` table.

The user and tenant information has to be passed to the `interceptRequests` advice. This is simply possible since information can be shared amongst several advices within the same aspect. Hence, the `getUserInfo` advice can store the user information in a variable within the `MTE` aspect, which is the used by the `interceptRequests` advice in the sense of Laddad's wormhole pattern [16].

Please note this advice is only specific to Tomcat but is independent of the application. Any other application server will require slight modifications of this advice.

Finally, we have to take care of tenant isolation. Using another advice within `MTE`, we intercept every access to a database `Connection` and re-direct access to the tenant schema. For JDBC accesses, the advice looks as follows:

```
@Around("call(java.sql.Connection
               java.sql.DriverManager.getConnection(..)
          && !within(com.siemens.aspects.MTE)")
public Object interceptGetConnection
                              (final ProceedingJoinPoint jp) {
  get the user and tenant (stored locally in MTE);
  Connection con = (Connection) jp.proceed(jp.getArgs());
                    // original logic gets connection
  Statement stmt = con.createStatement();
  // switch to tenant's database/schema:
  stmt.execute("SET SCHEMA '" + tenant + "'");
  return con;
}
```

Every successive database operation will use the tenant schema, i.e., database. Indeed, an `@After` advice would have been sufficient here. However, `@Around` is more flexible to handle other databases with different concepts such as an explicit database name in the URL. Section 5 will dive into the details and will also illustrate how to implement other strategies such as sharing the original tables between several tenants.

4.6 Customization

Several papers like [29] emphasize the importance of tenant-specific customizations of an application for business, thereby considering customization as a major challenge of multi-tenancy. Again, AspectJ can be used to give an application a tenant-specific behavior without explicitly touching the source code. To this end, each tenant-specific behavior requires one dedicated aspect, e.g., `TenantAModifier` for TenantA, which defines the specific tenant behavior. Since the logic of the aspect is technically applied to the overall application, the aspect must determine the expected tenant and only apply the logic to that tenant. That is why the aspect has to implement an interface `GenericModifier`, which demands for a method `getTenantName()`; it should return the tenant name of the modifier, i.e., "TenantA" for aspect `TenantAModifier`. Using `getTenantName()`, an advice can then compare the calling tenant with the expected one and modify the logic only for that tenant:

```
if (nameOfCallingTenant.equals(getTenantName()) {
  … modify logic …
} else { // don't modify behavior
   return jp.proceed(jp.getArgs()); // original logic
}
```

An `@Around` advice can define pointcuts where to modify logic. Inside the advice, the original call can be ignored by omitting `jp.proceed()`. Hence, functionality can be disabled, for example, by returning an empty result, a result masked out with stars '*', or an HTTP code 403 (FORBIDDEN) in case of REST services. Similarly, the original logic can be modified or extended. Especially information to be returned can be changed by using the original logic.

Adding new REST services offering additional functionality that is *not* part of the original application is more complicated since the logic will be implemented in a different class. We have to use static introduction to this end in the following manner:

```
@Aspect public class TenantAModifier
                                implements GenericModifier {
  @DeclareParents(
       defaultImpl=com.siemens.newfunc.NewFunction.class,
       value="com.siemens.app.ExistingAppl")
  public com.siemens.nf.NewFunctionIF mix;
}
```

Then, a new GET service `/newFunctionality`, can be implemented in the class `NewFunction`.

```
@Path("newFunctionality")
public class NewFunction implements NewFunctionIF {
  @GET public Response svcNewGetOperation(...) { ... }
}
```

The new logic implemented in `svcNewGetOperation` becomes available in class `ExistingAppl` (implementing the original REST service) because `Exist-ingAppl` inherits from the newly introduced superclass `NewFunction` – although its definition is done in another class. This happens because `@DeclareParents` places a new superclass `NewFunction` of interface `NewFunctionIF` on top of those classes that are specified by the `value` clause, here the single class `ExistingAppl`. The interface `NewFunctionIF` is only required for enabling a syntactic cast from `ExistingAppl` to `NewFunction`; the variable `mix` is of no further importance.

Thanks to AspectJ, the application itself does not have to be prepared or modified for allowing intercepted code at the right place. The powerfulness certainly depends on the power of the pointcut syntax and the context information available at the intercepted join points. The approach suffers only if certain points in the code cannot be addressed by pointcuts. Moreover, the application code has to be available to find appropriate join points; the weaving itself does not require the source code and is satisfied with byte code! This point is the major advantage of our approach: Other customization approaches require special, prepared customization points, where to plug in tenant logic. However, this would violate our goal not to touch the original application.

4.7 Monitoring

Every SaaS provider has to define a billing model for charging his tenants for using the application. In turn, a SaaS provider has expenses for running the application, especially in a public cloud. Then, he has to pay for the all used resources. In fact, the billing model must be appropriate to make profit. The investment covers both the operational costs in a Cloud as well as the costs for developing an application or SaaS-enabling it [23] and later maintenance [2].

Many proposed billing models for SaaS are post-paid. Tenants receive a bill and pays for usage periodically. Hence, the SaaS provider has to monitor and aggregate the consumption costs for each tenant [27] for billing purposes. If a SaaS provider charges his tenants by a fixed rate per month or based upon other factors such as the number of users (registered or in parallel), then it is important to throttle exhaustive usage by a single tenant because the SaaS providers' revenue will be reduced or even lost otherwise.

Consequently, it is necessary to monitor and log the activities of all tenants' users and the costs they produce. As [28] discusses, such a tracking is the task of the SaaS providers. The support given by underlying Cloud platforms is only rudimentary and not detailed enough to determine the costs for resources for each tenant individually.

To enable a tenant-specific monitoring, we have added the following table to the `Admin` schema in order to track tenants' user activities as shown in Table 2.

Table 2. Table UserMonitoring.

id	name	tenant	operation	timestamp	elapsed
1	UserA1	TenantA	Operation1	2016-11-10 17:00:01	12 ms
2	UserA2	TenantA	Operation2	2016-11-10 17:00:02	21 ms
3	UserB1	TenantB	Operation2	2016-11-10 17:00:03	10 ms

We again use AspectJ to intercept any user actions (maybe filtering out a few relevant ones by a pointcut). To this end, we extend the `interceptRequests` advice from Subsect. 4.5 to compute the elapsed time around `jp.proceed()`:

```
long start = System.nanoTime();
Object o = jp.proceed(args);
double elapsed = (double) (System.nanoTime() - start);
createLogEntry(user, tenant, elapsed,
               jp.getSignature().toShortString());
```

`createLogEntry` logs the elapsed time together with the signature of the method, tenant, user etc. at a central place. Dedicated pointcuts can define what has to be tracked; this might depend on the application. The table now gives an overview over all user activities and forms the basis for several scenarios. Using the table, tenants can be charged back for their consumed resources. Moreover, it is possible to check profit-making, i.e., whether the chosen billing model for one/all tenant(s) is appropriate to make profit. Also the (elapsed) execution times or the number of service requests for each user or tenant can be accumulated; if thresholds are exceeded, further access is throttled or rejected. Hence, a SaaS provider is able to timely react on frequent and massively active tenants by throttling them before costs rise. Even further use cases can be supported. For example, if a Service Level Agreement (SLA) specifies a maximum number of concurrent users, a `@Before` advice is able to check the current number of concurrent users for a tenant in the `UserMonitoring` table before executing a service request. Similarly, if an SLA states a threshold for the number of registered users, the `Users` table can be used to supervise the limit in the user registration process. Finally, all the monitoring information might be used to implement auto-scaling features that enable Cloud elasticity.

4.8 Configuration

AspectJ load-time weaving requires an additional configuration file aop.xml that specifies what aspects (`<aspects>`) are active and what packages (`<include …>`) should be intercepted by the aspect logic. The following content is an example:

```
<aspectj>
  <aspects>
    <aspect name="com.siemens.aspects.MTE"/>
    <aspect name="com.siemens.aspects.TenantAModifier"/>
    <aspect name="com.siemens.aspects.TenantBModifier"/>
  </aspects>
  <weaver> <include within="com.siemens.app.*"/>
  </weaver>
</aspectj>
```

5 Other Types of Database Servers and Isolation Strategies

We want to expand the scope of our investigation and discuss what has to be done to apply the principle to other database servers beside Oracle and to other data isolation strategies following Chong et al. [6]. This paper was one of the first to investigate multi-tenant data architectures and distinguishes between “separate databases”, “shared database, separate schemas”, and “shared database, shared schema” for an SQL Server. This section follows this structure and also investigates PostgreSQL and SQL Server databases as further candidates.

5.1 Separate Databases

Storing the tenant’s data in a separate database offers the highest degree of data isolation. In principle, using a separate database server for each tenant is an even higher isolation. For a strong isolation it is important to authenticate units for each tenant individually. Hence, there is essentially no difference between using a separate database server (i.e., an instance, containing several databases) and a database (within such a database server) as far as individual privileges can be defined for the units.

Oracle has a notion of a “database”, but this is closely related to a database server, named instance: Each instance can only be associated with one database. Hence, we have to set up an instance for each tenant. An Oracle JDBC URL thus refers to the instance as `jdbc:oracle:thin:@<Host>:1521:<Instance>`. This means for multi-tenancy that we have to replace `<Instance>` in the URL with the respective tenant’s instance name in order to switch from the existing instance to the tenant one. To this end, the pointcut for the `interceptGetConnection` advice from Subsect. 4.5 can still be used with minor changes of the advice:

```
@Around("call(java.sql.Connection java.sql.DriverManager
        .getConnection(String, String, String))
&& args(url,usr,pw) && !within(com.siemens.aspects.MTE)")
public Object interceptGetConnection(ProceedingJoinPoint
                 jp, String url, String usr, String pw) {
  get the user and tenant (stored in MTE);
  url = exchange instance name with tenant name in URL;
  usr = tenant;     // provide credentials
  pw  = password;   //  for database connect
  return jp.proceed(url,usr,pw);
}
```

PostgreSQL has both options, one DB server or one database for each tenant; the URL `jdbc:postgresql://<Host>:<Port>/<Database>` specifies the host, the database, and optionally the port number (if several PostgreSQL instances run on the same host):. The tenant-specific substitutions can be done analogous to Oracle in the `interceptGetConnection` advice using the same pointcut:

```
  url = replace host or database with tenant name in URL;
```

The URL of the SQL Server resembles the PostgreSQL URL: `jdbc:sqlserver://<Host>\<ServerName>;databaseName=<Db>`, i.e., server name and database name can be specified in addition to the host name. Hence, it is possible to change the DB server and/or database for corresponding tenants by a URL modification. Again, the same pointcut can be used with slight modifications of the related advice.

To sum up, only the advice has to be adapted to handle the different formats of URLs while keeping the original pointcut.

Furthermore, the Tenant Administration Service must create a tenant-specific database or server when a tenant has been approved. Here, the implementation depends on DB-specific concepts and dialects. Moreover, a user for the tenant with a password is required to provide database access in the `MTE` aspect. However, it is quite easy to organize the syntactic variants in Java. For example, users are created in Oracle with

```
CREATE USER TenantA IDENTIFIED BY <Pw>;
```

while PostgreSQL requires

```
CREATE ROLE TenantA WITH LOGIN ENCRYPTED PASSWORD '<Pw>',
                    TEMPLATE applicationTables;
CREATE DATABASE TenantA WITH OWNER=TenantA;
```

and SQLServer a statement like

```
CREATE USER TenantA IDENTIFIED BY <Pw>;
```

In general, the database set up of the original application has to be understood, i.e., what databases and schemas are available, which tables are tenant-specific etc. According to that, the creation of tables can be done during setup (cf. Subsect. 4.3) by executing `createApplicationTables.sql` in the tenant instance. PostgreSQL has a so-called template mechanism for handling the pre-creation of tables, views, stored procedures etc. during database creation (see the statement above).

5.2 Separate Schemas

A schema is basically a special concept of some database servers. The idea of a schema is to have a dedicated and isolated space within a database, e.g., one for each tenant. In such a schema, the same set of tables etc. can be created. Sometimes, even individual users and privileges can be specified for a schema.

In order to take care of data isolation, we can use the same pointcut `interceptGetConnection` as before in Subsect. 5.1 to intercept the request of a database connection for all types of database systems. The corresponding advice does not need to change the URL, but simply switches the schema by a statement that uses a database-specific syntactic variant. That is, all tenants connect to the same database setting the schema afterwards. How to use the Oracle schema for multi-tenancy has already been demonstrated in the `interceptGetConnection` advice in Sect. 4.5:

```
SET SCHEMA TenantA;
```

Furthermore, the Tenant Administration Service (cf. Sect. 4.4) has to create a schema for each tenant:

```
CREATE USER TenantA IDENTIFIED BY <Pw>;
```

As already mentioned in Subsect. 4.1, each Oracle user possesses a schema with the same name. Hence, there is no explicit schema definition.

PostgreSQL uses a different advice for the same pointcut due to a different syntax:

```
SET SEARCH_PATH TO "TenantA";
```

Again, the Tenant Administration Service has to create a tenant-specific schema, when a tenant has been approved:

```
CREATE ROLE TenantA WITH LOGIN ENCRYPTED PASSWORD '<Pw>'
CREATE SCHEMA "TenantAschema" AUTHORIZATION TenantA
```

SQL Server has a different syntax to switch the schema, too:

```
ALTER USER TenantA WITH DEFAULT_SCHEMA = TenantAschema;
```

User and schema have to be created in the Tenant Administration Service:

```
CREATE USER TenantA ...;
CREATE SCHEMA TenantAschema AUTHORIZATION TenantA;
```

The creation of tables can be done during setup by executing `createApplicationTables.sql` in the particular tenant schema.

Please note there is a strong danger of SQL injection in any case if a user can issue SQL arbitrary statements: a user can switch to another tenant's schema! Special prevention is required to prevent SQL injection.

If a database server does not support a dedicated schema concept, all the tenant-specific tables can be replicated in the same database by adding a tenant suffix: `<Table>_TenantA`. However, more effort is required because all SQL statements are affected due to changing table names in all queries. Furhermore, authentication with user/password is lost compared to an explicit schema concept. The principles of the next subsection can be applied to provide a smarter solution.

5.3 Shared Schemas

A third approach uses the same database and the same set of tables for all the customers. Thus, each table contains the data of several tenants. This certainly requires a discriminator column in each tenant-specific table, the `TenantId`. As an immediate consequence, every INSERT on such a table has to provide this `TenantId`, while queries (including delete and updates) must filter for the tenant's id by `TenantId=<`*`id`*`>`.

Indeed, this is the lowest level of data isolation, which requires the highest effort for application development. Anyway, AspectJ is able to handle the new arising challenges in a modular manner with general principles.

As a presumption, all tenant-specific tables must be known and handled. However, the `createApplicationTables.sql` script can be maintained, since `ALTER TABLE` statements can add the new `TenantId` column afterwards.

In contrast to the previous strategies, the Tenant Administration Service is not affected by this isolation strategy as no tenant-specific database or schema is required. In general, there is no dependency on the type of database server.

As already mentioned, all queries issued by the application have to be changed at runtime to add a filter `TenantId=<`*id*`>` . The interception is not an issue and can be done by pointcuts, e.g., for SQL queries executed by `executeQuery` in JDBC:

```
@Around("call(java.sql.ResultSet
                 java.sql.Statement.executeQuery(String))
    && !within(com.siemens.aspects.MultitenancyEnabler)")
public Object interceptExecute(ProceedingJoinPoint jp) {
  String theQuery = (String)jp.getArgs()[0];
  theQuery = "modified query";
  return jp.proceed(jp.getArgs());
}
```

The pointcut `interceptGetConnection` used in Subsects. 5.1 and 5.2 is no longer necessary. The principle is easy, but disguises a lot of technical issues. For example, let us assume the following original SQL query with two tenant-specific tables `Tab1` and `Tab2`:

```
SELECT *
FROM Tab1 t1 LEFT OUTER JOIN Tab2 t2 ON t1.id=t2.fk
WHERE t1.col1=10 OR t2.col2=20
```

Simply adding "`AND TenantId = <`*id*`>`" does not work since `TenantId` is ambiguous due to the two tables with a `TenantId` column. Even an addition "`AND t1.TenantId=<`*id*`>AND t2.TenantId=<`*id*`>`" is incorrect because of `OR` in the `WHERE` clause, which changes the original semantics drastically. Obviously, brackets are required around the `OR` condition. Next, `LEFT OUTER JOINs` must be treated carefully. A condition "`(t1.col1=10 OR t2.col2=20) AND t1.TenantId=<`*id*`>AND t2.TenantId=<`*id*`>`" returns wrong results in many database servers as such a condition in the `WHERE` clause diminishes the outer join by implicitly forcing a join. Finally, `SELECT *` has additional `TenantId` columns for which an existing cursor is not prepared. In sum, the correct form is:

```
SELECT concrete columns without TenantId
FROM Tab1 t1 LEFT OUTER JOIN Tab2 t2 ON t1.id=t2.fk
   AND t1.TenantId=<id>AND t2.TenantId=<id>
WHERE (t1.col1=10 OR t2.col2=20)
AND t1.TenantId=<id>
```

This simple example already shows some important pitfalls. Further points to be handled appropriately are inner queries with `IN` and `EXISTS`. Hence a lot query string parsing and manipulation is required for queries, making the logic in the advice quite complex. This has to be done for `DELETE` and `UPDATE` statements in the same manner. Furthermore, `INSERT` statements have to be modified, too, because of the new `TenantId` column. Here, the `TenantId` value has to be added to the `VALUES` clause.

One nasty challenge are stored procedures. Again, it is easy to intercept the invocation of stored procedures in JDBC. But the source code of those procedures is not directly accessible because of being stored in the database. The solution we suggested is as follows:

- Modify the code of all the stored procedures manually, i.e., modifying the SQL statements according to the previous discussion (this does not require a re-compilation of the application code);
- add a new parameter TenantId to each procedure in order to transport the tenant information to the procedure;
- modify the procedure call by passing the TenantId as a parameter to the procedure call during interception.

So far, the use of JDBC for database accesses has been discussed. Hence, the question arises what happens if an object/relational (O/R) framework such as Hibernate or EclipseLink is used. There are two general options:

- To intercept JDBC at a deeper level, i.e., inside the O/R framework or within a connection pool. Please remember that AspectJ is able to intercept even 3rd party libraries without having source code available. The pointcuts do not need to be changed, however, the packages to be intercepted are specific to a particular framework and must be listed in the aop.xml configuration file (cf. Subsect. 4.8).
- As an alternative, the higher O/R requests can be intercepted by pointcuts. Different String modifications then become necessary for query languages such as JPQL of the JPA standard. This is more complex since the new TenantId properties must be added to the persistent Java classes to match the changed table structures with the TenanId column. This might have an impact on the build process and/or Java code.

6 Evaluation

6.1 Modularity and Adaptability

Separation of concerns is one of the driving forces of aspect oriented programming. Bringing in the notion of reuse without compromising the advantages of separation of concerns is an important consideration for application development. The positive impact of code reuse during application development and maintenance should not be under-weighed.

The strategies to enable multi-tenancy follow the best practices of using AspectJ and the individual concepts can be reused across other similar applications. All the multi-tenancy logic is concentrated in classes to be added to the application's WAR file thereby adjusting the aop.xml configuration file accordingly. Moreover, tenant specific logic is also clearly separated in particular classes. Only a restart of Tomcat is required for the multi-tenancy configuration to take effect.

All the multi-tenancy components rely on simple mechanisms that can easily be applied to other legacy Java applications to make them multi-tenant. Thus, development cost can be reduced for other applications. Indeed, REST services are easier to handle than applications with a graphical user interface since there are pure Java methods annotated with `@GET`, `@PUT` etc., which are the entry points for functionality. Anyway, background logic of other applications can be handled the same way.

The `MTE` aspect that takes care of tenant isolation mainly depends on tools, i.e., the application server and the database server, especially the isolation strategy to apply. This aspect has to be adapted if `MTE` should be applied to applications using JBoss and/or MySQL, for instance. Sticking to the same technologies allows for an immediate reuse of the `MTE` aspect. The pointcuts to intercept DB accesses rely on JDBC or an object/relational framework and are not DB-specific. Hence, only switching the persistence technology requires a modification of pointcuts.

However, the pointcut `interceptRequests` in `MTE` depends on the application methods to be intercepted just as customization does; other applications require different pointcuts and/or advices.

Anyway, any adaption and modification is made in central components – outside the original application. Reusability can be further enhanced. An abstract aspect can implement an advice but leaves out the pointcut, while application-specific sub-aspects reuse the general logic and only specify the concrete pointcuts.

6.2 Implementation Effort

Taking a look at the lines of code, the simplicity of the approach becomes obvious:

- The new Tenant Administration Service has about 400 lines of Java code;
- The aspect `MTE` consists of ca. 150 lines all together for the Oracle schema approach, however, a shared-schema approach requires about 500 lines of code due to a more complex logic;
- The effort for a customizing `TenantXModifier` aspect depends on what should be modified. To give an impression, disabling functionality in a REST service requires 10 lines, a simple modification of service behavior 23 lines, and introducing a new REST service about 60 lines.

6.3 Lessons Learned

The lack of comprehension and maintainability of aspect-orientation is often criticized. Since we only have a small number of dedicated aspects serving a very special purpose such as tenant isolation, customization, and monitoring, we did not detect any problems in this respect. Indeed, the impact of multi-tenant aspects to behavior is clearly arranged.

As explained throughout the paper, we could benefit a lot from aspect-orientation to achieve our goal to leave code untouched. Especially, the possibility to intercept 3-rd-party tools such as Tomcat and to exchange information between advices according to the “Wormhole Pattern” [16] helped a lot.

However, we also recognized some limitations. The first idea was to have a `Users` table in each tenant schema instead of global table. As a consequence, Tomcat authentication has to use the corresponding tenant database. However, we failed to intercept the start-up of Tomcat to bring in the logic. That is the reason why the approach relies on the single `Users` table of the existing application.

6.4 Advantages

We achieve with our AspectJ approach the general advantages of full multi-tenancy such as cost saving by sharing resources (hardware, application server, database etc.) amongst tenants and reducing operational expenses (OPEX). But the major additional advantage of our approach lies in the fact that the source code of the existing application does not need to be touched explicitly.

In fact, Tenant Administration Service (cf. Sect. 4.4) is just a new service to be deployed in the Tomcat application server as a WAR file. Tenant isolation is achieved by adding a new `MTE.class` to the deployed application WAR. Additional files Tenant*X*Modifier.class in the WAR provide a tenant-specific behavior for each Tenant*X*. Only a restart of Tomcat is required to apply the `MTE` aspect thanks to AspectJ load-time weaving. Adding a new tenant class can even be done at runtime without a restart by just deploying the Tenant*X*Modifier class and adjusting the aop.xml file.

Hence, the approach offers a cost-efficient way to speed up time-to-market by migrating existing applications quickly into SaaS-offerings. The approach also allows for a flexible configuration, e.g., for various tenant isolation strategies (one DB for each tenant, one schema for each tenant, or one single-table for all tenants).

7 Conclusions

While research has investigated many facets of multi-tenancy for designing and implementing new applications, this paper focuses on migrating legacy single-tenant to fully multi-tenant applications. This is an important and necessary step to offer an existing application as Software-as-a-Service (SaaS).

There are a couple of approaches and methodologies that demonstrate how to convert legacy applications into multi-tenant software. However, they require to re-engineer the legacy source code to a large extent. In contrast, our approach consists of simply adding components to the legacy application – *without* explicitly touching the application’s source code.

We propose several components, being implemented as aspects in AspectJ, which have to be added to an application’s WAR file. The major component for tenant isolation depends only on technological choices such as application server, database server, and the chosen data isolation strategy. Furthermore, tenant customization depends on the application; pointcuts specify what to intercept in the application and advices implement the customization.

In order to validate the approach, we used an existing industrial REST application that runs in Tomcat and uses an Oracle database. In particular, we discuss how to achieve three main concerns in detail:

- tenant isolation [5] for different strategies and database servers;
- tenant-specific customization of behavior;
- monitoring tenants' user activities for billing purposes.

We elaborated upon how to benefit from the aspect-oriented language AspectJ in order to achieve these points. We presented the AspectJ approach in detail and evaluated the approach with regard to modularity, adaptability, and implementation effort. The effort to be spent for the overall principle requires only a few 100 lines of aspect code. We also concluded with some lessons learnt. The approach can directly be adapted to other Java applications, especially REST services.

In general, REST services are easier to handle than applications with a graphical user interface since there is pure Java code without any parts in HTML or Javascript. In order to evaluate the limits, our future work will consider applications with a graphical user interface. First experiences show that the MTE (Multi-Tenancy Enabler) aspect works well for achieving data isolation. Moreover, logic can be customized on a per-tenant basis as far as no GUI is concerned. Further investigations are required to evaluate customizing the UI.

Currently, applying the presented aspects to other applications requires some copy&paste of code and an adjustment of pointcuts and advice logic. This is also true for the MTE aspect which depends on technologies. Feature modelling tools might be useful to generate the aspect code according to a domain-specific language that describes the database server, data isolation strategy, and application server. Alternatively or in addition, we think of providing a reusable aspect framework. The idea is to have an aspect hierarchy that reflects technological choices. If for example an application uses JBoss and SQLServer with a shared schema isolation approach, then an application-specific sub-aspect has to derive from an Oracle_JBoss_SharedSchema aspect. The sub-aspect itself only contains those parts that are specific to the application, e.g., the database URL and pointcuts.

In case of too much load, several Tomcat instances have to be started with a load balancer in front. Hence, migrating an application into the cloud is much more than just adding multi-tenancy. Taking care of scalability issues and replacing software components with Cloud services is also subject to future work.

References

1. Andrikopoulos, V., Binz, T., Leymann, F., Strauch, S.: How to adapt applications for the Cloud environment - Challenges and solutions in migrating applications to the cloud. Computing **95**(6), 493–535 (2013)

2. Bezemer, C., Zaidman, A. Platzbeecke, B. Hurkmans, T., Hart, A.: Enabling multitenancy: an industrial experience report. In: Technical Report of Delft University of Technology, TUD-SERG-2010-030 (2010)
3. Bezemer, C., Zaidman, A.: Challenges of reengineering into multitenant SaaS applications. In: Technical Report of Delft University of Technology, TUD-SERG-2010-012 (2010)
4. Binz, T., Leymann, F., Schumm, D.: CMotion: a framework for migration of applications into and between clouds. In: SOCA 2011, pp. 1–4 (2011)
5. Chong, F., Carraro, G.: Architecture strategies for catching the long tail (2006). https://msdn.microsoft.com/en-us/library/aa479069.aspx. Accessed Nov 2016
6. Chong, F., Carraro, G., Wolter, R.: Multi-tenant data architecture (2006). http://msdn.microsoft.com/en-us/library/aa479086.aspx. Accessed Nov 2016
7. Elrad, T., Filman, R., Bader, A. (eds.): Theme section on aspect-oriented programming. CACM **44**(10) (2001)
8. Fehling, C., Leymann, F., Mietzner, R.: A framework for optimized distribution of tenants in cloud applications. In: IEEE 3rd International Conference on Cloud Computing (CLOUD), pp. 252–259 (2010)
9. Guo, C., Sun, W., Huang, Y., Wang, Z., Gao, B.: A framework for native multi-tenancy application development and management. In: CEC/EEE 2007: International Conference on Enterprise Computing, E-Commerce Technology and International Conference on Enterprise Computing, E-Commerce and E-Services, pp. 551–558 (2007)
10. Hohenstein, U., Koka, P.: An approach to add multi-tenancy to existing applications. In: ICSOFT 2016, pp. 39–49 (2016)
11. Khajeh-Hosseini, A., Greenwood, D., Smith, J., Sommerville, I.: The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise. Softw. Pract. Exp. **42**(4), 447–465 (2012)
12. Kiczales, G., et al.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, pp. 230–242 (2007)
13. Kong, L., Li, Q., Zheng, X.: A novel model supporting customization sharing in SaaS applications. In: International Conference on Multimedia Information Networking and Security (MINES), pp. 225–229 (2010)
14. Krebs, R., Momm, C., Kounev, S.: Architectural concerns in multi-tenant SaaS applications. In: CLOSER 2012, pp. 426–431 (2012)
15. Kwok, T., Nguyen, T., Lam, L.: A software as a service with multi-tenancy support for an electronic contract management application. In: International Conference on Services Computing (SCC), pp. 179–186 (2008)
16. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming, 2nd edn. Manning, Greenwich (2009)
17. Lee, W., Choi, M.: A multi-tenant web application framework for SaaS. In: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 970–971 (2012)
18. Lee, J., Kang, S., Hur, S.: Web-based development framework for customizing java-based business logic of SaaS application. In: 14th International Conference on Advanced Communication Technology (ICACT), pp. 1310–1313 (2012)
19. Li, Q., Liu, S., Pan, Y.: A cooperative construction approach for SaaS applications. In: 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 398–403 (2012)
20. Lizhen, C., Haiyang, W., Lin, J., Pu, H.: Customization modeling based on metagraph for multi-tenant applications. In: 5th International Conference on Pervasive Computing and Applications (ICPCA), pp. 255–260 (2010)

21. Mell, P., Grance, T.: The NIST definition of cloud computing. National Institute of Standards and Technology, September 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Accessed Nov 2016
22. Moens, H., Truyen, E., Walraven, S., Joosen, W., Dhoedt, B., De Turck, F.: Developing and managing customizable software as a service using feature model conversion. In: IEEE Network Operations and Management Symposium (NOMS), pp. 1295–1302 (2012)
23. Momm, C., Krebs, R.: A qualitative discussion of different approaches for implementing multi-tenant SaaS offerings. In: Proceeding Software Engineering 2011, pp. 139–150 (2011)
24. Orue-Echevarria, L., et al.: Cloudifying applications with ARTIST: a global modernization approach to move applications onto the cloud. In: CLOSER 2014, pp. 737–745 (2014)
25. Park, J., Moon, M., Yeom, K.: Variability modeling to develop flexible service-oriented applications. J. Syst. Sci. Syst. Eng. **20**(2), 193–216 (2011)
26. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
27. Ruiz-Agundez, I., Penya, Y., Bringas, P.: A flexible accounting model for cloud computing. In: SRII 2011, pp. 277–284 (2011)
28. Schwanengel, A., Hohenstein, U.: Challenges with tenant-specific cost determination in multi-tenant applications. In: 4th International Conference on Cloud Computing, Grids and Virtualization 2013, pp. 36–42 (2013)
29. Shahin, A., Samir, A., Khamis, A.: An aspect-oriented approach for SaaS application customization. In: 48th Conference on Statistics, Computer Science and Operations Research 2013, Cairo University, Egypt, pp. 1–15 (2013)
30. Tsai, W., Shao, Q., Li, W.: OIC: ontology-based intelligent customization framework for SaaS. In: IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–8 (2010)
31. Tsai, W., Sun, X.: SaaS multi-tenant application customization. In: IEEE 7th International Symposium on Service Oriented System Engineering (SOSE), pp. 1–12 (2013)
32. Walraven, S., Truyen, E., Joosen, W.: A middleware layer for flexible and cost-efficient multi-tenant applications. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 370–389. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25821-3_19
33. Wang Z. et al.: A study and performance evaluation of the multi-tenant data tier design pattern for service oriented computing. In: IEEE International Conference on eBusiness Engineering, (ICEBE), pp. 94–101 (2008)
34. Wang, H., Zheng, Z.: Software architecture driven configurability of multi-tenant SaaS application. In: Wang, F.L., Gong, Z., Luo, X., Lei, J. (eds.) WISM 2010. LNCS, vol. 6318, pp. 418–424. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16515-3_52
35. Zhou, X., Yi, L., Liu, Y.: A collaborative requirement elicitation technique for SaaS applications. In: 2011 IEEE International Conference on Service Operations, Logistics, and Informatics (SOLI), pp. 83–88 (2011)