An Approach to Add Multi-tenancy to Existing Applications

Uwe Hohenstein and Preeti Koka
Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, D-81730 Munich, Germany

Keywords: Multi-tenancy, Cloud Migration, Aspect-Orientation, AspectJ, Industrial Application, Case Study.

Abstract: Multi-tenancy, i.e., sharing resources amongst several tenants, is a key element to make SaaS profitable by saving resources and operational costs. This paper considers multi-tenancy in the context of Cloud migration and presents an approach to let existing applications become multi-tenant. The novelty of this approach is that no reengineering and modification of the application’s source code is required. Adding some new components is sufficient to achieve tenant management, authentication, tenant isolation, and also customization. Using a case study, the paper demonstrates in detail how to benefit from aspect-orientation, particularly the AspectJ language, in this respect and concludes with experiences.

1 INTRODUCTION

According to (Mell and Grance, 2011), Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources that can rapidly be delivered with a minimal management effort or service provider interaction.

Particularly, software is more and more becoming an on-demand service drawn from the Cloud. The so-called Software-as-a-Service (SaaS) is a delivery model that enables customers, the so-called tenants, to lease services without a local installation (Lee and Choi, 2012). Tenants pay for what they use to what extent without buying software licenses.

While a traditional application service provider typically manages one dedicated application instance per tenant, SaaS providers usually adopt a multi-tenant architecture (Chong and Carraro, 2006). Multi-tenancy is a software architecture principle that lets several tenants share a common infrastructure. This saves operational cost due to an efficient utilization of hardware and software resources and improved ease of maintenance (Bezemer and Zaidman, 2010). A well-economical SaaS application has to pursue a multi-tenant architecture.

In order to benefit from features such as elasticity and pay-as-you-go, businesses want to move applications into the cloud. One challenge for industry is to convert legacy applications into multi-tenant SaaS without major code changes and high effort (Binz et al., 2011), thus preserving investments while entering SaaS business.

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 – without refactoring 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. Currently, each tenant has an application deployed on a Tomcat server and an Oracle database on premise.

Several authors such as (Walraven et al., 2011) or (Guo et al., 2007) 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., (Andrikopoulos et al., 2013) define further degrees of sharing and categorize other 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 our application multi-tenant with lowest development effort is certainly to let each tenant obtain a VM with Tomcat, the application, and Oracle. (Krebs et al., 2012) call this a virtualization approach. The ease of this approach is paid by well-known disadvantages such as high consumption of resources and high costs especially in public clouds. Moreover, each tenant requires an Oracle license or additional costs for using Oracle as a Cloud service.
Fully efficient multi-tenancy (Chong and Carra- 
ro, 2006), at the other edge of the scale, allows for 
sharing all resources, one Tomcat, one application, 
and one Oracle server among all tenants. Setting up 
a fully multi-tenant application requires a significant 
re-engineering of applications, and thus causes high 
development costs (Momm and Krebs, 2011).

While serving several tenants by sharing one in-
stance, SaaS applications have to be customizable or 
configurable to fulfill the varying functional re-
eruiresments of individual tenants (Krebs et al., 2012). 
Tenant customization is recognized as one important 
requirement and challenge by (Guo et al., 2007), 
(Bezemer et al., 2010), and others. (Lee and Choi, 
2012) state 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 (Pohl et al., 2005) to offer 
variability. Using aspect-oriented programming 
(AOP) is sometimes proposed to achieve configur-
bility, e.g., by (Shahin et al., 2013) and (Wang and 
Zheng, 2010).

In this paper, we also apply AOP, more precisely 
the AspectJ language (Laddad, 2009), to migrate 
existing applications into fully multi-tenant SaaS 
applications. We elaborate upon how to benefit from 
AspectJ in this context and show that it is possible to 
have a simple and cheap mechanism by only adding 
components to existing applications – without any 
urther reengineering. Using a real existing industrial 
application, we demonstrate the major advantages of 
our migration approach. In a nutshell, it is possible to 
achieve tenant isolation, to modify existing be-
havior 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.

The motivation for our work is manifold. The 
approach is a first step to let existing applications 
become Cloud ready and to enable entering SaaS 
business fast and easily. Such a first trial can explore 
SaaS business opportunities 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 ef-
fort. And finally, free demo versions of existing 
applications can be made publicly available in a 
Cloud as a teaser, maybe with reduced functionality. 
Since no profit can be directly made in that case, we 
benefit from small investments in development.

The remainder of this paper is structured as fol-
lows. Section 2 presents related research and deduc-
tes the necessity for this work.

Before discussing the migration approach, we 
give in Section 3 a short introduction into the aspect-
oriented AspectJ language, as far as it is necessary to 
understand how we applied AspectJ.

We used a concrete industrial project to prove ef-
effectiveness. Section 4 introduces the application 
in its original single-tenant form and presents our ap-
proach to migrate to a multi-tenant Cloud applica-
tion with low programming effort in detail. We dis-
cuss the components that implement important fac-
et of multi-tenancy such as tenant isolation and 
customization.

In Section 5, we evaluate the AspectJ approach 
and discuss the lessons learned. Finally, the conclu-
sions summarize the discussion and presents future 
ideas.

2 RELATED WORK

Several papers are related to our work in two sepa-
rate directions: Multi-tenancy and cloud migration.
The possible variants of multi-tenancy are de-
scribed, among others, by (Chong et al., 2006) and 
(Kwok et al., 2008). (Momm and Krebs, 2011) con-
sider approaches to reduce resource consumption 
and discuss some cost aspects of sharing. Based on 
the number of tenants, the number of users per ten-
ant, and the amount of data per tenant, (Wang et al., 
2008) make recommendations on the best multi-
tenant variant to use.

(Guo et al., 2007) discuss implementation princi-
iples for application-level multi-tenancy, exploring 
different approaches to improve isolation of security, 
performance, availability, and administration.

(Fehling et al., 2010) come up with prospects for 
the optimization of multi-tenancy by distributing the 
tenants with respect to Quality of Service.

An architectural approach for reengineering appli-
cations to enable multi-tenancy in software services is 
defined by (Bezemer et al., 2010). Their multi-
tenancy reengineering pattern requires a multi-tenant 
database, tenant-specific authentication, and configu-
ration. The discussion takes into multi-tenancy reen-
gineering account workflow and UI configuration. 
The reengineering pattern is applied to an existing 
single-tenant application. Because of a well-designed 
and layered architecture, the effort was relatively 
little. Furthermore, (Bezemer and Zaidman, 2010) 
manually transform the ScrewTurn wiki case to a 
multi-tenant application and encounter performance 
isoation of tenants, scalability issues for tenants from
different continents, security, data protection, configurability, and data isolation as the core challenges, however, not solving all these issues. The paper also stresses on another cost aspect for multi-tenant applications: The recurrence of maintenance tasks (e.g., patches or updates) raises operating cost too.

In contrast to this general work on strategies and their impact on resource consumption, we tackle the problem of adding multi-tenancy in a smart way.

A lot of research considers tenant-specific customizations as an important requirement. Case studies such as (Kwok et al., 2008) stress on configurability of multi-tenant applications. (Tsai et al., 2010) discuss the elements of an application that need to be customized: Graphical user interface, workflow (business logic), service selection and configuration, and data.

According to (Shahin et al., 2013), customization could be performed in two ways. In a source-code manner, SaaS applications are customized by integrating new tenant-specific source code. (Zhou et al., 2011) and (Kong et al., 2010) pursue such an approach. Despite providing tenants with flexibility in the customization process, this approach suffers from several drawbacks. The tenant must be aware of the implementation details of the SaaS application. Next, allowing tenants to integrate source code may violate the security regulations of the application. And finally, the process of software upgrades becomes more complicated for the SaaS provider, since all the tenant-specific extensions have to be retained.

In an alternative composition-based approach, SaaS applications are customized by composing variants, selected from a provided set of components. The current state of practice in SaaS development is that configuration of pre-defined extensions is preferred over source code based approaches, which is considered too complex as pointed out in (Walraven et al., 2011). One prominent approach is providing an application template with unspecified parts, often called customization points (Lizhen et al., 2010), which can be configured by selecting predefined components from a provided set (Moens et al., 2012); (Park et al., 2011); (Li et al., 2012).

(Pohl et al., 2005) point out four key concerns to be addressed: modelling customization points and variations, describing relationships among variations, validating customizations performed by tenants, and dis-/associating variations to/from customization points during runtime.

(Shahin et al., 2013) deal with all these concerns. Illustrated by a Travel Agency example, they propose the Orthogonal Variability Modeling (OVM) to model customization points and variations and to describe the relationships among variations. A Metagraph-based algorithm validates tenants’ customizations. 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.

(Lizhen et al., 2010) deal with three of the above concerns by using Metagraphs to model customization points, variants, and their relationships. They also propose an algorithm to validate customizations made by tenants.

(Tsai et al., 2010) and (Tsai and Sun, 2013) handle only the modeling of customization points and variants using an ontology-based customization framework with OVM. To avoid unpredictable customizations, tenants are guided through the customization process.

(Walraven et al., 2011) consider middleware component models as inflexible for offering software variations to different tenants. They use Google AppEngine to build a multi-tenancy support layer that combines dependency injection with middleware support. Using an online booking scenario, they evaluate operational expenses and flexibility. The application requires dedicated customization points for applying customization.

(Wang and Zheng, 2010) apply the more general AOP in a case study, but still have to prepare the software architecture accordingly.

All this research starts from green-field or requires at least a preparation of an existing application with customization points, while our approach leaves the original application as it is.

Another branch of related research considers migrating legacy applications into the cloud as a challenge. There is a lot of work on checklists and methodologies to perform such migrations. The project ARTIST (Orue-Echevarria et al., 2014), e.g., provides methods, techniques, and tools to guide companies to move applications into the cloud. A methodology supports the complex, time-consuming, and expensive transition with three phases: pre-migration, migration, and post-migration.

Vendor lock-in is seen by (Binz et al., 2011) as a major difficulty for migrating existing applications into and between different clouds. The CMotion framework models entities and their dependencies to support migration, but requires the implementation of adapters.

(Khajeh-Hosseini et al., 2012) present a framework to support decision makers. Decisions to migrate existing systems to the cloud can be complicated as the benefits, risks, and costs of using the
Cloud are complex and should also consider organizational and socio-technical factors. Their Cloud Adoption Toolkit offers a collection of tools for decision support and helps to identify those concerns and match them to appropriate technologies. A cost modeling tool is presented in detail with a case study; the tool can be used to compare the cost of different cloud providers and deployment options.

To our knowledge, there is no work combining an approach to migrate applications to the Cloud with adding multi-tenancy for a real 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 (Kiczales et al., 1997). AO provides systematic means for effective modularization of crosscutting concerns (CCCs), i.e., those functionalities that are typically spread across several places in the source code. CCCs often lead to lower programming productivity, poor quality and traceability, and lower degree of code reuse (Elrad, 2001).

AO has brought up new languages with special concepts to modularize CCCs and to avoid the well-known symptoms of non-modularization such as code tangling and code scattering.

The AspectJ language (Laddad, 2009) essentially extends Java with aspects. An aspect can change the dynamic structure of a program by intercepting certain points of the program flow, called join points. Examples of join points are method and constructor calls or executions, field accesses, and exceptions. Join points are syntactically specified by means of pointcuts. Pointcuts identify join points in the flow by means of a signature expression. Advices specify certain actions to be taken before and/or after the join points. The following is an example for a simple AspectJ aspect:

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

An annotation @Aspect lets the Java class MyAspect become an aspect. A method annotated with @Before, @After, @Around is an advice that is executed before, after or around join points. Those annotations specify pointcuts as a string. MyAspect possesses a before advice that adds logic before executing those methods that are captured by the pointcut myPc: 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. “+” as a type stands for any type. Parameter types can be fixed or left open (..).

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. In addition to aspect annotations, AspectJ offers a language of its own (in fact, an extension of Java), however, requiring an AspectJ compiler. But a compiler changes the build process, which is often not desired, so for us. We do not want to re-compile the existing application.

4 ADDING MULTI-TENANCY TO EXISTING APPLICATIONS

4.1 Application Case

Our case study considers an existing Java application, a REST service in the travel management domain with the following characteristics.

- The application is currently shipped as a single-tenant application to customers and deployed in Tomcat at the customer site.
- Each customer obtains a full application stack of its own, consisting of Tomcat, the application, and an Oracle database (DB) server at the backend for storing data, all running on-premise.

The goal is to deploy this application in a public cloud while sharing Tomcat, the application and the database amongst several tenants.

Tomcat offers several forms of user authentication: form-based (for Web application), basic authentication (for REST services) etc. If authentication is enabled, user and password are requested for accessing an application. User, passwords, and user roles are maintained in a configurable “user/roles store” like in an XML file, a relational DB, a JDNi store etc. During authentication, the Tomcat container checks for access control privileges against that store. In addition, the application can restrict functionality to users with specific roles. Our application uses the basic authentication of Tomcat.

An Oracle database possesses a dedicated schema Auth(entication) that contains the user/roles tables. The connect string with a specific user/password is part of the Tomcat configuration file.

Oracle has very specific terms that should be un-
derstood. Any access to a database has to be granted to users. Each user has a password to login and an associated DB schema with the same name. To access data of another user, tables can be prefixed by a schema name. However, a user has to be explicitly granted access by the owner, i.e., schemas are isolated from each other by default. Each user can create the same set of tables with the same statement – in his schema. Thus, the concept of a tenant “database” maps to a user/schema within one Oracle server. 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 DB server with exactly one database being associated. A JDBC driver connects to that database.

Details about the application are subject to confidentiality and irrelevant for the message of this paper.

4.2 Tenant and User Management

The major concern of this paper is to let an existing Tomcat application become multi-tenant, thus sharing a Tomcat and the Oracle database instance. As pointed out in (Bezem et al., 2010), the prerequisite for multi-tenancy is an appropriate tenant/user management supporting the following workflow:

1. There must be a possibility to let tenants register for using the application, if they are interested.
2. After having clarified the payment details between the SaaS provider and the tenant and set up a contract, a SaaS administrator should be able to acknowledge or deny the tenant for using the application. To this end, we set up a new organizational Tomcat role saas which allows the administrator to manage SaaS. After acknowledgement, each tenant obtains a Oracle user and schema, i.e., a DB of its own thereby keeping the tenant’s application data isolated.
3. The SaaS provider usually delegates the tasks of creating and maintaining users to each tenant. Any acknowledged tenant obtains a TAdmin role that allows registering his users for the application by specifying credentials for authentication.
4. The tenant’s users obtain a role User and can log in to the application once they are authenticated. This principle is called a centralized authentication system in (Chong and Carraro, 2006).

Thus, we have the following roles in Tomcat giving privileges to the various types of users: the administrator for SaaS applications (saas), the administrator for a tenant who is enabled to register tenant’s users (TAdmin), and the user of the application (User); indeed, there may be several with specific privileges. For the ease of discussion, we collapse them to one.

4.3 Initial DB Setup for Multi-tenancy

We assume a database schema Appl in the original application to keep the application’s data (indeed, there might be several). Another schema, referred to as Auth, contains the tables Users and User_Roles to keep Tomcat users with their roles. Tomcat accesses these tables for authentication to check the password and roles. This means that only Tomcat users in the Users table are allowed to access the application, provided they have the requested role.

First, we extend the Users table in the Auth schema with a column tenant to keep the association between a user and the tenant s/he belongs to.

We add a new Oracle user/schema admin exclusively used by the SaaS admin to keep information about tenants. A new table tenants is created in this schema to keep registered tenants with their administrators. We also add a UserMonitoring table to the schema for monitoring purposes (cf. 4.7).

A new Tomcat user saas with a role saas is added to the Users and User_Roles table. This allows him to use the new tenant administration (cf. 4.4).

Finally, we need an SQL script createApplicationTables.sql that can be executed in any new tenant schema to add the application’s tables.

These steps can be done by means of an SQL script, without affecting the existing application. Other Tomcat authentication schemes require similar steps, e.g., operating on XML files.

4.4 Tenant Administration Service

We need new services for administration purposes, especially for registering tenants and users, and new functionality according to the workflow in Subsection 4.2. These services can simply be deployed as a new application in Tomcat in order to become immediately effective. There is again no impact on the existing application source code. For instance, a REST server can offer the following major services supporting the workflow.

1. POST TenantService allows tenants to register for the application. The payload specifies a name Tenant, an administrator name, and a password, both being used for Tomcat authentication. This information is stored in the table Admin.Tenants (name, admin, password, acknowledged, ...). Access to this service is granted to everyone.

2. PUT TenantService/Tenants/[TenantId] with a body {“acknowledge”:Yes|No} can only be used by
the SaaS administrator to enable or disable access for TenantA. If a TenantA is acknowledged by the administrator, then acknowledged=1 is set for TenantA in the Admin.Tenants table. The record for TenantA’s admin is taken from Admin.Tenants and added to the Auth.Users and Auth.User_Roles tables, assigning a Tomcat role TAdmin. Next, a new schema TenantA is created with all the application tables for exclusively keeping application data for TenantA, executing the SQL script createApplicationTables.sql.

(3) POST TenantService/Tenants/{Tenant} creates a user for TenantA to make the user known to the application. Using the payload, a user name and a password are added to the Auth.Users table for Tomcat authentication giving the users a User role; the association of a user to his tenant is stored in the tenant column of the Auth.Users table. This service is only accessible by the registered tenant admin, i.e., the role TAdmin is checked by Tomcat authentication. If the administrator AdminA for TenantA registers two users UserA1 and UserA2, the Users and User_Roles tables finally contain the contents shown in Table 1; an explanation describes when each record has been added.

<table>
<thead>
<tr>
<th>Users</th>
<th>user_name</th>
<th>user_pass</th>
<th>tenant</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ ex. users</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>SaaS</td>
<td>SaaS</td>
<td>NULL</td>
<td>TenantA</td>
</tr>
<tr>
<td>AdminA</td>
<td>PwA</td>
<td>TenantA</td>
<td></td>
</tr>
<tr>
<td>UserA1</td>
<td>PwA1</td>
<td>TenantA</td>
<td></td>
</tr>
<tr>
<td>UserA2</td>
<td>PwA2</td>
<td>TenantA</td>
<td></td>
</tr>
</tbody>
</table>

(4) UserA1 and UserA2, registered for TenantA, are now able to authenticate. They are allowed to use the application with a Tomcat User role.

### 4.5 Making an Existing Application Become Tenant-aware

Tenants and their users are now known to the Tomcat application. Indeed, all these users are allowed to access the application since Tomcat authenticates against the Auth.User/UserRoles tables. Moreover, the application still uses the existing tables in the App1 schema for all users. Hence, the overall effect is as if the application has new users, but without any effective data isolation for tenants.

However, the application must enforce measures to ensure isolation between different tenants (Guo et al., 2007). To achieve data isolation, a user’s data must be stored in the tenant’s schema (i.e., database). This requires determining the tenant for a logged-in user to use the correct schema; any access must be directed to that one. Here, AspectJ comes into play to intercept every authentication: the user is determined and the corresponding Tenant for the user is derived in such a way that the original application is not explicitly modified, i.e., compiled and/or rebuilt. This is the novelty in our approach.

The following code sketches an AspectJ @Around advice which changes the behavior accordingly:

```java
@Aspect public class MTE {
    @Around("execution(* com.siemens.app.ExistingAppl.svc*(..))
        & !within(com.siemens.app.aspects.MTE)"
    public Object interceptServices (final ProceedingJoinPoint jp) {
        (1) determine user from HttpServletRequest and derive role & tenant (from Users table);
        (2) store user/tenant/role for later usage;
        return jp.proceed(jp.getArgs()); /* call original logic of svc* method */
    }
}
```

We use Java with AspectJ annotations and LTW instead of the AspectJ language and compiler. The annotation @Aspect makes a Java class MTE (Multi-tenancyEnabler) be an aspect. The annotation @Around defines an advice to be executed at join points. @Around includes a pointcut as a String to determine the relevant join points: Any execution of methods starting with svc... belonging to the basic REST service class ExistingAppl returning a value of any type (*) with any parameters (...). Instead of wildcards, we could also specify several method signatures individually and combine them with ‘||’.

The @Around method interceptServices implements the logic to be executed at each join point. This advice traps the execution of svc... methods and replaces the behavior with its body. The parameter jp of type ProceedingJoinPoint serves two purposes. First, it is used to execute the original logic at the join points by means of jp.proceed(). Furthermore, jp gives access to the context of invocation such as the parameter values (jp.getArgs()) and the signature of the join point (jp.getSignature()), i.e., the concrete svc... method to be executed.

Since the method is implicitly invoked in our aspect inside by jp.proceed(), we must exclude this invocation in order to avoid an endless loop. That is the reason why !within(MTE) is added in the pointcut...
to not intercept any invocation that occurs within the aspect itself.

Only the pointcut "execution(*
com.siemens.app. ExistingAppl.svc{(..)}" of interceptServices, which specifies what methods or services to intercept, depends on the application code, here the class ExistingAppl that implements the REST service.

All of this looks very straightforward, but with one challenge: how to get the user name from Tomcat authentication (cf. (1) in the code above)?

Usually, there is a HttpServletRequest req, which can be used to derive authentication information, e.g., by req.getUserPrincipal().getName(). Such a variable declaration can be annotated with @Context in a class; the value is then injected by the Tomcat container. But there are also other ways that could have been used in the original application, e.g., passing an additional @Context HttpServletRequest parameter to a service method. Unfortunately, it is unknown what mechanism has been used, and moreover, even a global variable req is usually private and not accessible from an external aspect.

We noticed that Tomcat invokes for authentication in any case a _handleRequest method of a class WebApplicationImpl. A @Before advice can pick up this information in the MTE aspect, to give access to the ServletRequest and the user name:

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

Please note AspectJ intercepts JARs, even of 3rd party tools; the unavailability of the source code does not hinder AspectJ to intercept Tomcat!

The clause this(w) binds the variable w to the called object of type WebApplicationImpl. The method getThreadLocalHttpContext() is used to get the user who has logged in. The tenant, the user belongs to, can be determined by using the Auth.Users table.

But now the next challenge (2) arises: How can the advice interceptServices access this information? This is possible because the aspect can be used for sharing information. The advice getUserInfo can store the user information in a variable within the MTE aspect. The advice interceptServices simply uses this information. That is, the user information is shared amongst several advices in the sense of Laddad's wormhole pattern (Laddad, 2009).

This advice does not depend on the application but only on Tomcat. Any other application server will require slight modifications of this advice.

Next, for the purpose of tenant isolation, any connection to the database requested from the application must be directed to the tenant schema. This is achieved by another advice within MTE, e.g.:

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

In case of Oracle, we have to set the tenant’s schema for any successive DB operation. Indeed, an @After advice would have been sufficient here. However, other databases have different concepts to use such as an explicit database name in the URL. This partially requires to modify the getConnection parameters (obtained by jp.getArgs()) in the advice accordingly before calling the original logic.

We can also implement other strategies such as sharing the original tables between several tenants in this advice. Obviously, this advice depends on the database system and the isolation strategy, but not on the application.

### 4.6 Customization

According to (Shahin et al., 2013) and others, a tenant-specific customization of an application is a major challenge of multi-tenancy. Once the tenant is known, AspectJ enables the SaaS provider to give an application a tenant-specific behavior without changing the existing source code.

From an implementation point of view, each tenant-specific behavior requires one aspect class. Such a class has to implement an interface TenantModifier which demands for a method getTenantName(). A class TenantAModifier implements GenericModifier could define an aspect for TenantA. Using the getTenantName method, an advice can compare the calling tenant with the expected TenantA and modify the logic only for that tenant:

```java
if (nameOfCallingTenant.equals(getTenantName())) {
    // modify logic ...
```
We first show how to add new features to a specific tenant, offering additional functionality that is not part of the original application, e.g., adding new REST services. Please note this is not possible by simply adding the same @path to another class in a new WAR; Tomcat will complain. However, static introduction in AspectJ helps here.

```java
@Aspect
public class TenantAModifier

    implements GenericModifier {

        @DeclareParents defultImpl=com.siemens.

        .newfunc.NewFunctionalityIF class,

        value="com.siemens.app.ExistingApp1"

        public com.siemens.nf.NewFunctionalityIF mix;

    }
```

@DeclareParents adds a new superclass NewFunctionalityIF on top of those classes that are specified by the value clause, here the single class ExistingApp1, which implements the REST service. The new functionality, e.g., a new GET service /newFunc, can then be implemented in the class NewFunctionality.

```java
@Path("newFunc")
public class NewFunctionality

    implements NewFunctionalityIF {

        @GET public Response svcNewGetOperation(...) {

            ...
        }
```

The new service will be available in the ExistingApp although its definition is done in another class. The interface NewFunctionalityIF has only syntactical reasons to enable a cast from ExistingApp to NewFunctionality, its “new” superclass. The variable mix is of no further importance.

Existing features can be disabled by putting an advice around a pointcut that catches the method, then ignoring the original call by omitting `jp.proceed()`. An empty result, a result masked out with stars ‘*’, or an HTTP code 403 (FORBIDDEN) in case of REST services can also be returned.

Similarly, an @Around advice can modify the existing behavior, for example extending or changing information to be returned, removing some records or fields from a result etc. In any case, the original logic and result can be used for modifications.

One important question is what has to be prepared by the application in order to allow intercepted code at the right place. The power of what can be achieved depends on the pointcut syntax (specifying what to intercept, i.e., where to replace logic) and the context information available at those join points. Intercepting methods is sufficient for REST services and always feasible. Anyway, the application code has to be known to find appropriate join points (although the aspect itself is satisfied with byte code and does not require the source code). This is in contrast to other customization approaches that require prepared customization points, where to plug in tenant logic, thus violating our goal to leave the original application untouched.

### 4.7 Monitoring

The next point is related to economical concerns of SaaS providers. On the one hand, IaaS/PaaS providers define cost models; a SaaS provider has to pay for running an application in a public cloud. On the other hand, a SaaS provider has to define a billing model to charge his tenants for using the application. Both models have to be balanced in such a way that a SaaS provider is able 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 (Momn and Krebs, 2011) and later maintenance (Bezemer et al., 2010).

Most popular public cloud billing models are post-paid models where the tenant receives a bill and pays for usage periodically. This requires monitoring and aggregating the consumption costs of each tenant (Ruiz-Agundez et al., 2011). A SaaS provider can also charge his tenants by a fixed rate, e.g., per month or based upon other factors such as the number of users (registered or in parallel). Here, it is important to predict the costs a tenant’s usage will produce. Moreover, exhaustive usage by one tenant could reduce the SaaS providers’ revenue.

No matter what billing model would be applied, it is necessary to monitor and log the activities of all tenants’ users and the costs they produce.

(Schwanengel and Hohenstein, 2013) discuss the challenges of calculating the costs each tenant generates in a public cloud to establish a profitable billing model for a SaaS application. They show that only rudimentary support is available by cloud providers. A user receives a monthly bill from a cloud provider, not being detailed enough to determine the costs for resources for each tenant individually.

To enable a tenant-specific monitoring, we added a table UserMonitoring(id, name, tenant, operation, timestamp, elapsed) to the Admin schema in order to track tenants’ users activities.

We again use AspectJ to intercept any relevant user actions. To this end, we extend the interceptServices advice in Subsection 4.5 to compute the elapsed time around `jp.proceed()` and to log it 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. This forms the basis for

- a consumption-based model that charges back tenants for their consumed resources;
- a tenant-specific profit-making check, i.e., whether the chosen business model for one/all tenant(s) is appropriate to make profit;
- a timely reaction on frequent and massively active tenants to throttle them before costs rise.

The presented approach is quite general and can support further use cases. If a Service Level Agreement (SLA) specifies a certain maximum number of concurrent users, this SLA can be checked by a Before advice: Before executing a service request, the number of concurrent users for the tenant is checked in the UserMonitoring table. Similarly, it is possible to accumulate the (elapsed) execution times or the number of service requests for each user or tenant and to throttle or reject further access if thresholds are exceeded. 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, we can use the monitoring information to implement auto-scaling features that benefit from a Cloud’s elasticity.

5 EVALUATION

5.1 Advantages

We certainly achieve the general advantages of multi-tenancy such as cost saving by sharing resources (hardware, application server, database etc.) among tenants and reducing operational expenses (OPEX).

The additional advantage of our approach lies in the fact that the source code of the existing application does not need to be modified explicitly.

In order to add tenant management, we have to deploy a new admin service (cf. 4.4) as a WAR file. Adding TenantXModifier.class to the deployed application WAR, we achieve tenant isolation. Additional files TenantXModifier.class in the WAR can provide a tenant-specific behavior for each TenantX. This facilitates customization. Thanks to AspectJ load-time weaving, only a restart of Tomcat is required to make the multi-tenancy aspects apply.

All the multi-tenancy logic is concentrated in classes to be added. These components rely on simple mechanisms that can easily be applied to other legacy applications to make them multi-tenant. Thus, development cost can be reduced.

The MTE aspect mainly depends on tools, i.e., the application server and the DB system, especially the isolation strategy to apply. That is, a modification of this aspect becomes necessary only if we, e.g., switch to JBoss and/or MySQL. The pointcuts to intercept DB accesses are not DB-specific but rely on JDBC. Any adoption and modification to other tools can be made in central components anyway.

However, the pointcut interceptServices in MTE also specifies the application methods to be intercepted. Other applications require different pointcuts. To this end, an abstract aspect can implement an advice, but leave out the pointcut, while application-specific sub-aspects specify the concrete specific pointcuts, but reuse the general logic.

In general, REST services are easier to handle. There are methods annotated with @GET, @PUT etc., which are the entry points for functionality. Moreover, it is just Java code.

Taking a look at the lines of code, it becomes obvious how simple the approach is:

- The new tenant management service consists of about 400 lines of Java code;
- The aspect MTE has about 150 lines;
- Any TenantXModifier certainly depends on what should be modified. To give an impression for REST services, an advice to disable functionality has 10 lines, an advice introducing a new REST service about 60, and an advice for a simple modification of behavior has 23 lines.

To sum up, the approach offers a cost-efficient way to migrate existing applications quickly and cheaply into SaaS-offerings speeding up time-to-market. Moreover, our approach also allows for a flexible configuration, e.g., for tenant isolation (one DB server or one DB for each tenant, single-table for all tenants).

5.2 Lessons Learned

In case of AOP, comprehension and maintainability are often cited. However, we did not detect any problems. In fact, we only have a small number of aspects. Moreover, the behavior of aspects is manageable, especially since AO is not available to all programmers.

One major contribution of AOP we benefit from is the possibility to exchange information between even unrelated classes by means of an aspect according to the “Wormhole Pattern” (Laddad, 2009).

Anyway, we also detected some limitations: The first idea to let newly registered users to authenticate was to have a users table in each tenant schema.
Authentication must then be intercepted to refer to the correct database. We ran into the technical problem that authentication is part of the Tomcat container, i.e., to intercept loading the application is not sufficient. Our attempts to intercept loading Tomcat failed (but maybe we overlooked a possibility).

6 CONCLUSIONS

This paper presented an approach for migrating existing single-tenant to fully multi-tenant applications, which can afterwards be offered as SaaS.

While other approaches require a more or less large reengineering of the existing source code in order to bring in multi-tenancy, our approach consists of simply adding components to the legacy application — without explicitly touching the existing application’s source code.

The additive components are implemented as aspects in AspectJ and depend only on technological choices such as application server, database system, and the data isolation strategy. The components can easily be applied to other applications by just adjusting some pointcuts.

Using an existing REST application that runs in Tomcat and uses Oracle, the paper presented the approach and discussed how to achieve two main concerns in detail:
- tenant isolation (Chong and Carraro, 2006);
- tenant-specific customization of behavior.

We demonstrated the details of the approach and the effort to be spent; the overall principle requires only a few 100 lines of aspect code. We also elaborated upon how to benefit from AspectJ in this respect and concluded with some evaluation and lessons learnt.

In fact, REST services are simpler to handle as demonstrated in this paper. There is just Java code without any parts in HTML or Javascript. Future work will go one step further and tackle other types of applications, especially to determine the limits. So far, we have first experiences that show the true aspect working well. Customization of logic is also possible as far as no GUI is concerned. Further investigations will handle customizing the UI. Moreover, we want to combine our approach with feature modeling tools to control the configuration in an easy manner.

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

REFERENCES


Laddad, R., 2009: AspectJ in Action: Practical Aspect-


Tsai, W., Shao, Q., Li, W., 2010. OIC: Ontology-based intelligent customization framework for SaaS. In IEEE Int. Conf. on Service-Oriented Computing and Applications (SOCA), 2010, pp. 1–8.