# A Middleware Layer for Flexible and Cost-efficient Multi-Tenant Applications

Stefan Walraven, Eddy Truyen, and Wouter Joosen

IBBT-DistriNet, Katholieke Universiteit Leuven
3001 Leuven, Belgium
{stefan.walraven,eddy.truyen,wouter.joosen}@cs.kuleuven.be

**Abstract.** Application-level multi-tenancy is an architectural design principle for Software-as-a-Service applications to enable the hosting of multiple customers (or tenants) by a single application instance. Despite the operational cost and maintenance benefits of application-level multi-tenancy, the current middleware component models for multi-tenant application design are inflexible with respect to providing different software variations to different customers.
In this paper we show that this limitation can be solved by a multi-tenancy support layer that combines dependency injection with middleware support for tenant data isolation. Dependency injection enables injecting different software variations on a per tenant basis, while dedicated middleware support facilitates the separation of data and configuration metadata between tenants. We implemented a prototype on top of Google App Engine and we evaluated by means of a case study that the improved flexibility of our approach has little impact on operational costs and upfront application engineering costs.

**Keywords:** Multi-tenancy, Dependency injection, Software-as-a-Service, Google App Engine

## 1 Introduction

**Context.** An important trend in the landscape of service-oriented software has been the rise of the "Software-as-a-Service" (SaaS) delivery model [31] where software applications are created and sold as highly configurable web services. A well-known SaaS provider delivers for instance a Customer Relationship Management (CRM) application [28] as a configurable service to a variety of customers that each have their specific preferences and required configurations.

SaaS applications differ from traditional application service provisioning (ASP) in the sense that economies of scale play a much more important role. A traditional application service provider typically manages one dedicated application instance per customer. In contrast, SaaS providers typically adopt a *multi-tenant architecture* [7], meaning that a shared application instance hosts multiple customers, which are called tenants. The primary benefit of this approach is that the *operational costs can be significantly reduced*: (i) hardware and software resources can be more cost-efficiently divided and multiplexed across customers, and (ii) the overall maintenance effort is seriously simplified because upgrading the application software can be performed for all tenants at once.

**Problem.** Application-level multi-tenancy comes however also with a number of disadvantages. More specifically, in this paper we focus on two challenges when implementing multi-tenancy at the application level. First *application engineering complexity* is increased. The engineering of multi-tenant application software is more complex than traditional single-tenant applications that are deployed per individual tenant. The primary cause is that the application developer should take measures to ensure isolation between different tenants with respect to the application configuration and data of each tenant [15]. Moreover, a tenant-specific management facility needs to be created such that application configuration management per tenant is separated from the core application management by the SaaS provider.

Secondly, in order to meet the unique requirements of the different tenants, the application must be *highly configurable and customizable*. The current state of practice in SaaS development is that configuration [7,15] is preferred over customization which is considered too complex [30]. Configuration usually supports variance through setting pre-defined parameters for the data model, user interface and business rules of the application. Customization on the other hand involves software variations in the core of the SaaS application in order to address tenant-specific requirements that cannot be solved by means of configuration. Compared with configuration, customization is currently a much more costly approach for SaaS vendors because it introduces an additional layer of application engineering complexity and additional maintenance overhead.

**Approach & Contribution.** This paper presents a software development and execution platform[1] for building and deploying customizable multi-tenant applications, narrowing down the gap between configuration and customization. More specifically, we present a multi-tenant middleware layer on top of Platform-as-a-Service (PaaS) platforms that (i) supports improved customization flexibility, (ii) preserves the operational cost benefits of the application-level multi-tenancy principle, and (iii) frees the application developer from a lot of initial application engineering costs for multi-tenancy.

We implement our middleware layer on top of Google App Engine (GAE) [13]. We extend the Guice dependency injection framework [14] with support for tenant-specific activation of software variations and use the scalable and high-performance datastore of GAE for storing and isolating tenant-specific application metadata. We evaluate the feasibility of our middleware layer by comparing a standard single-tenant and multi-tenant application with a flexible version that is developed using our middleware layer. This shows that the impact of our middleware layer on operational costs and additional application engineering complexity is minimal.

**Structure of the Paper.** The remainder of this paper is structured as follows. Section 2 introduces the case study and motivates the need for a middleware

[1] Other aspects of SaaS applications such as SLA management, metering and billing are out of the scope of this paper.

that supports true application-level multi-tenancy with improved customization flexibility. Subsequently, Section 3 presents the architecture of our middleware layer and its implementation on top of Google App Engine. Section 4 presents the evaluation of our middleware architecture in the three dimensions of customization flexibility, operational costs, and initial engineering costs. Section 5 elaborates on related work and Section 6 concludes the paper.

## 2 Problem Elaboration & Motivation

This section first explores the design space of multi-tenant applications and positions our intended middleware architecture in this space. Subsequently our work is motivated by means of an application case. Finally, the main requirements for our middleware layer are derived from a customization scenario in this application case.

### 2.1 Multi-tenancy Architectural Strategies

Multi-tenancy aims to maximize resource sharing among customers of a SaaS application and to reduce operational costs. However different architectural strategies can be applied to achieve multi-tenancy. As shown in Fig. 1, multi-tenancy can be realized at the application level, middleware level or virtualized infrastructure level. Each approach makes a different trade-off between (i) minimizing operational costs (including infrastructural resources as well as maintenance cost), (ii) minimizing upfront application (re-)engineering costs, and (iii) maximizing flexibility to meet different customer requirements.

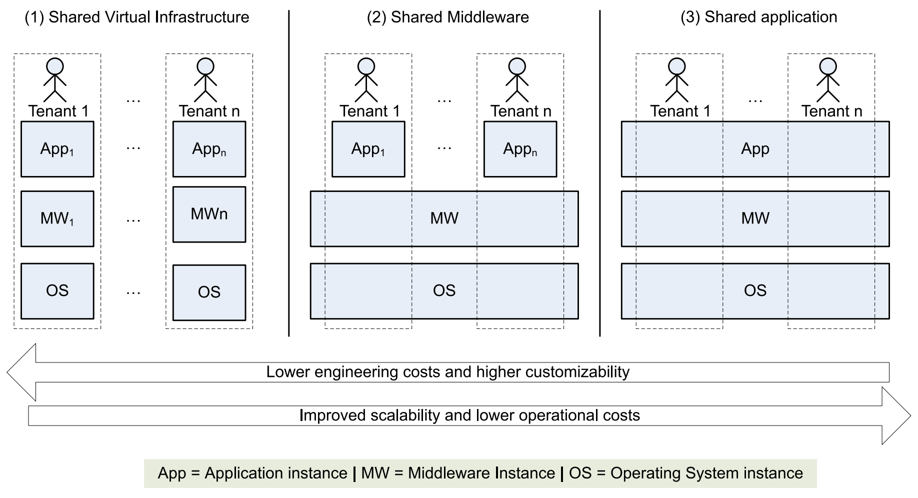
(1) Shared Virtual Infrastructure
(2) Shared Middleware
(3) Shared application
Tenant 1
Tenant n
App₁
Appₙ
App
MW₁
MWn
MW
OS
Lower engineering costs and higher customizability
Improved scalability and lower operational costs
App = Application instance | MW = Middleware Instance | OS = Operating System instance

**Fig. 1.** Different architectural approaches to achieve multi-tenancy.

As stated in the introduction, application-level multi-tenancy maximizes the level of resource sharing but is also the least flexible choice with additional engineering overhead. At the other end of the spectrum, virtualization technology can be used to run multiple operating system partitions with dedicated application and middleware instances for each tenant on shared servers. The advantage of this approach is its increased flexibility and low upfront application engineering cost. However, fewer tenants can be hosted on a single server and maintaining separate application instances per tenant also has a much higher cost than with application-level multi-tenancy.

Middleware-level multi-tenancy [5,2] uses a separate middleware platform that is able to host multiple tenants on top of a shared operating system, which may be either placed on a physical or virtualized hardware. In this way, the initial engineering complexity for multi-tenancy is shifted from the application level to a reusable middleware layer that also offers basic support for isolation of tenants. However, the component and deployment model of these middleware architectures still require that a separate application instance is deployed for each tenant which again implies a higher maintenance cost.

Our proposal is to create middleware support for building true multi-tenant applications with the flexibility to adapt to tenant-specific requirements. Because all tenants are served by the same instance of the application, this means that there is need for tenant-specific software variability in the application components. We assume that such multi-tenant application components do not maintain tenant-specific state, but that all tenant-specific state is stored in a (separate) database. To ensure scalability when user load increases, a pool of identical application instances with our middleware layer have to be created. Existing PaaS platforms already take care of this scalability requirement in a transparent way. For example, Google App Engine automatically scales up (and down) by creating extra instances as the load increases. We therefore propose to incept our middleware layer as an extension for PaaS platforms.

## 2.2 Motivating Example

Consider the example of a SaaS provider for on-line hotel booking (see Fig. 2). The SaaS provider offers a highly configurable web service that travel agencies can use for booking hotels and flights on behalf of their customers. Travel agencies play in this example the role of tenant whereas employees and customers of a travel agency are considered the users that belong to a tenant. Employees are offered a customized user interface and customers of the travel agency can login to check the status of the travel items through a URL with a custom-made domain-name that corresponds with the travel agency. A special 'tenant administrator' role is assigned to someone who is responsible for configuring the SaaS application, setting up the application data and monitoring the overall service. This role can be played by an internal or external client of the SaaS provider or even resellers who are an intermediate business proxy. In the context of this simple example, the tenant administrator belongs to the ICT staff of a travel agency company.

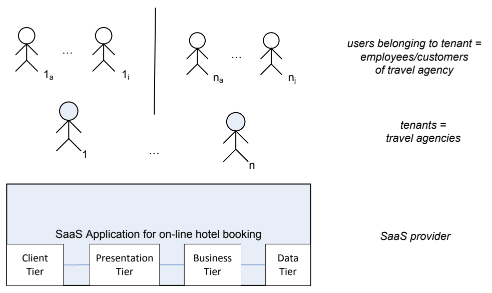
users belonging to tenant = employees/customers of travel agency
tenants = travel agencies
SaaS provider
SaaS Application for on-line hotel booking
Client Tier
Presentation Tier
Business Tier
Data Tier

**Fig. 2.** SaaS application for on-line hotel booking.

### 2.3 Requirements Derived from a Customization Scenario

Suppose that a particular travel agency wants to be able to offer price reductions to their returning customers. As such, the on-line hotel booking application should be extended with an additional service for managing customer profiles and a service for calculating price reductions. We assume that SaaS providers employ a business model where the base application is offered to tenants at no or low cost, but tenants incur an additional price for additional services. Based on this simple scenario, we can derive requirements with respect to core development, service customization and runtime support.

With respect to development, the application development team of the SaaS provider should be offered a simple way to *manage the different tenant-specific variations* as separate units of deployment that can be selectively bound to the core architecture of the application. Moreover, the overall 'multi-tenancy concern' should be well separated from the application layer.

With respect to customization, tenant administrators should be offered a *configuration facility* to select what software variations should be enabled for them (e.g. the price reduction service). In addition, this facility should also allow to specify specific configuration parameters (e.g. business rules for the price reduction service). This configuration data should be stored in the datastore of the SaaS provider in an isolated way under a specific tenant ID.

The runtime support of the middleware layer must provide support for *injecting software variations on a per tenant basis*. When a user (either customer or employee) logs in, the tenant to which the user belongs should be determined. Based on the acquired tenant ID, the multi-tenant middleware should then activate the appropriate software components to process the requests of the user. Another key requirement of the execution platform is that the tenant-specific software variations should be applied in an isolated way without affecting the service behavior that is delivered to other tenants.

## 3 Middleware Support for Tenant-specific Customization

This section presents the overall architecture of our middleware layer to support tenant-specific customization of SaaS applications. The component model of our middleware layer targets multi-tier applications and structures the application into a core architecture with declared variability points for multi-tenant software variations. Building on top of this component model, the middleware layer consists of a support layer for tenant administrators and run-time support for injecting software variations on a per tenant basis.

In this paper we focus on the customization of component-based multi-tier applications, rather than business processes (e.g. BPEL). The latter requires a different approach where software variations are deployed as separate services, and per tenant a separate business process is responsible for the coarse-grained composition of the appropriate services. In the context of component-based applications, dependency injection (DI) [11] is a common composition mechanism. With standard DI however, separate object hierarchies are maintained per tenant in a shared address space which increases heap memory storage and supports only static binding of software variations. Therefore, we prefer a composition mechanism that allows in situ run-time rebinding of variations. This requires an extension to the DI mechanism.

This section is structured as follows. We first propose an extension to the the multi-tier component model to make it tenant-aware. Next we describe in depth the architecture of our multi-tenancy support layer. Finally, the prototype implementation of this middleware layer on top of Google App Engine [13] is presented.

## 3.1 Tenant-Aware Component Model

To cope with the different and varying tenant requirements, we apply a *feature-based approach*. Software variations are then expressed in terms of features. A feature is a distinctive functionality, service, quality or characteristic of a software system or systems in a domain [17]. Ideally these features are modular software units that can be easily composed into the base application. As illustrated in Fig. 3, variation points are specified in the base application, representing the locations where features should be composed. A feature can have several alternative implementations (e.g. `I1` and `I2` in the figure). Based on the tenant-specific configuration, one of the feature implementations is bound to the variation points across the different tiers.

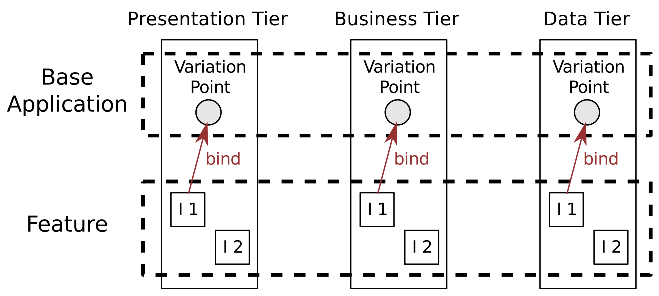

**Fig. 3.** Illustration of the feature-based approach.

Our extension to the component model supports the application developers of the SaaS provider to develop features as software modules. For each feature different implementations can be registered. A feature implementation consists of a set of software components (possibly at different tiers) and specifies how these components are bound to the base application. The concept of features is necessary to enable the SaaS provider to easily ensure the *consistency* of software variations across the different tiers of the SaaS application.

In addition, the developers need to be able to tag the locations in the base application where tenant-specific variation is allowed. To *annotate these variation points*, we introduce a new annotation: `@MultiTenant`. Listing 1 shows the annotation of a field with the price calculation service interface. This variation point initiates customization of the on-line hotel booking application based on the currently applicable tenant-specific configuration, for example price calculation with price reduction. Because a variation point can be bound by different features, the annotation has an optional parameter specifying the feature it belongs to. This enables developers to limit the variation point to a specific feature.

**Listing 1.** Annotation of a variation point for price calculations.

```
...
 @MultiTenant
 private IPriceCalculatorStrategy priceCalculatorStrategy;
...
```

## 3.2 Architecture of the Multi-tenancy Support Layer

The architecture of our middleware layer supporting flexible multi-tenant applications is presented in Fig. 4. This support layer consists of a *flexible middleware extension framework* to manage features, specify tenant-specific configurations and to dynamically activate the required variations on a per tenant basis via dependency injection. This approach relies on a *multi-tenancy enablement layer*, offering basic multi-tenancy support and facilitating the separation of data and configuration metadata. Our multi-tenancy support layer serves as an extension to middleware platforms, but especially to Platform-as-a-Service (PaaS) solutions. Possibly such a PaaS already offers built-in support for tenant data isolation.

**Multi-tenancy Enablement Layer.** The base for application-level multi-tenancy is isolation between the different tenants, such as isolation of data, performance and faults. To achieve tenant-specific customization the main requirement is isolation of data, more specifically configuration metadata. With the default single-tenant approach, the configuration of an application is specified in a global configuration file. In a multi-tenant context a global configuration file results in a uniform application for all tenants, preventing tenant-specific customization. Any change to the configuration would affect all tenants. Therefore tenant-specific configurations have to be stored separately and applied within the scope of a tenant, instead of globally.

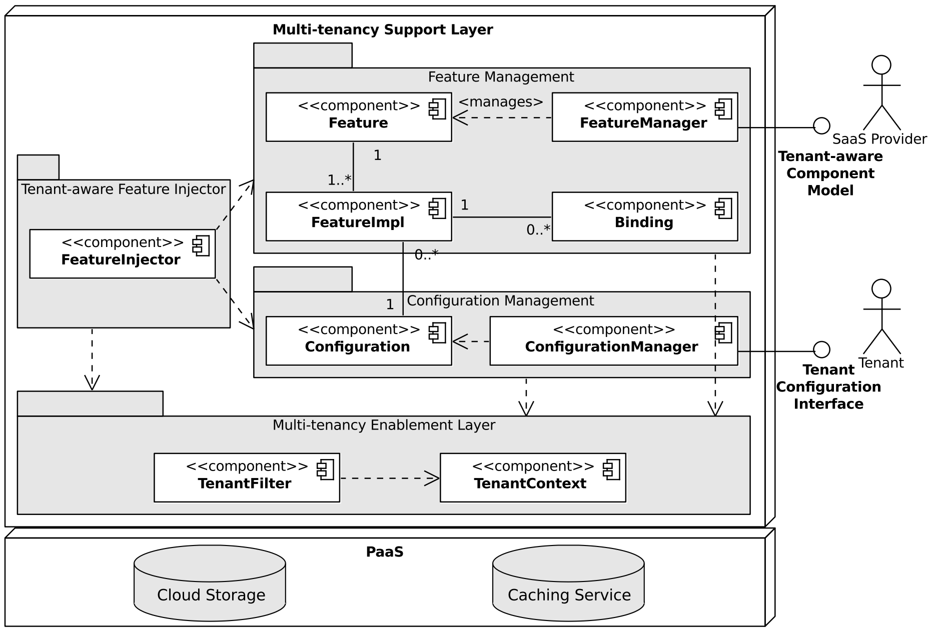
Multi-tenancy Support Layer
Feature Management
<<component>> Feature
<manages>
<<component>> FeatureManager
1
1..*
<<component>> FeatureImpl
1
0..*
<<component>> Binding
0..*
Tenant-aware Feature Injector
<<component>> FeatureInjector
Configuration Management
1
<<component>> Configuration
<<component>> ConfigurationManager
SaaS Provider
Tenant-aware Component Model
Tenant
Tenant Configuration Interface
Multi-tenancy Enablement Layer
<<component>> TenantFilter
<<component>> TenantContext
PaaS
Cloud Storage
Caching Service

**Fig. 4.** Overview of the multi-tenancy support layer.

To achieve tenant data isolation three main components are required: (i) the *tenant context* containing the information of the tenant linked to the current request (via a unique tenant ID), (ii) *tenant-specific authentication* to identify the tenant, and (iii) *multi-tenant data storage.* Incoming requests are filtered to retrieve the tenant ID (e.g. based on the request URL) and to set the current tenant context. Multi-tenant data storage can be obtained by applying filters that intercept the calls to the storage API and inject the tenant ID from the associated tenant context. In addition, comparable interceptors are necessary for the caching service (distributed in-memory storage). This allows to rapidly retrieve tenant-specific configurations, without large I/O performance overhead.

**Flexible Middleware Extension Framework.** The flexible middleware extension layer provides the following functionality:

1. a *feature management facility* providing an API to manage the variability of the application and the available feature implementations,
2. a *configuration management facility* to manage the default and tenant-specific configurations,
3. a *feature injector* to dynamically inject the required software variations conforming the tenant-specific configurations.

*Feature Management.* The `FeatureManager` manages the set of available features and their different implementations. A `Feature` specifies at least the following information: a unique identifier (e.g. feature name) and description for the feature, and the set of registered implementations for that feature.

A `FeatureImpl` contains the description of the feature implementation, a set of bindings, and a reference to the configuration interface of this implementation. Each `Binding` specifies the mapping from a variation point to a specific software component. This metadata about the features is globally accessible by both the SaaS provider and the tenants, and therefore should not be isolated. The `FeatureManager` offers a development API to enable the SaaS provider to create and register features and feature implementations, while the tenants are able to inspect the different features via the tenant configuration interface.

*Configuration Management.* Since a feature can have multiple implementations, each tenant can specify its preference for a specific feature implementation via the tenant configuration interface. Such a `Configuration` description defines the mapping from a feature to a specific feature implementation, more specifically from a feature ID to a `FeatureImpl`. The different tenant-specific configurations are then managed by the `ConfigurationManager`. In contrast to the feature descriptions, the tenant-specific configurations are stored on a per tenant basis.

Furthermore, the SaaS provider has to specify a configuration containing for each feature the mapping to a default feature implementation. If a tenant does not specify his tenant-specific configuration, this default configuration will be automatically selected.

*Tenant-aware Feature Injection.* Based on the features registered in the `Feature-Manager` and the default as well as tenant-specific configurations, our multi-tenancy support layer has to activate the appropriate feature implementations when required. To achieve this we apply the *dependency injection (DI)* pattern [11]. Instead of instantiating the feature implementations directly in the application, the flow of control is inverted: the life cycle management of feature implementations is controlled by a dependency injector or provider. This injector binds dependencies in the application to an implementation file. Such a *binding* is traditionally but not necessarily a mapping between a type (generally an interface or abstract class) and an implementation type (a class or component). This concept of a binding between a dependency and an implementation corresponds to our `Binding` between a variation point and a software component, as specified in the `FeatureImpl`s. As a result, in the above `ConfigurationManager` a tenant-specific configuration corresponds to a specific configuration of the DI framework.

For each variation point in the application the tenant-aware `FeatureInjec-tor` decides at runtime which implementation needs to be used, based on the configuration that applies. First, the `FeatureInjector` intercepts the requests to a dependency and consults the `ConfigurationManager`. The latter queries the multi-tenant data storage using the tenant ID to retrieve the tenant-specific configuration. Subsequently, the right binding is obtained from the `Configuration`, specifying the mapping between the variation point and a specific software component. This software component is instantiated and injected in the application to further handle the request. If the appropriate binding is not available in the tenant-specific configuration, the default configuration is used. In case the feature

ID parameter was given, the search to the appropriate binding can be narrowed down to the bindings of a specific feature implementation.

Finally the injected instance is stored in the cache in an isolated way using the tenant ID. For the following requests by this tenant that involve the same variation point, the `FeatureInjector` queries the cache. Using this tenant-aware caching service enables us to support flexible multi-tenant customization of a shared instance without the associated performance overhead.

### 3.3 Implementation

We implemented a prototype of our multi-tenancy support layer on top of Google App Engine (GAE) [13] (SDK 1.5.0), using the Java programming language and the Guice dependency injection framework [14] (v3.0). Google App Engine is a PaaS plaform to build and host traditional web applications developed with Java Servlets and Java Server Pages (JSP). GAE has built-in support for tenant data isolation via the Namespaces API. A separate namespace is assigned to each tenant. We only had to implement a `TenantFilter` to map incoming requests to a specific namespace and to configure that all requests have to go through this filter. For caching we use the Memcache service.

We chose Guice as DI framework because it is type-safe and compatible with GAE. However, it does not support the execution of tenant-specific injections: all dependencies are set globally. Any modification would affect all tenants. This is a general problem with dependency injection because it does not support activation scopes.

To solve this issue, we added an extra level of indirection. Instead of injecting features, we inject a `Provider` for that feature. This way the servlets have a dependency to a provider of a feature instead of to the feature itself. This generic `FeatureProvider` decides based on the tenant-specific configuration which feature implementation should be selected. However, the customizations that can be performed this way are limited to switching between implementations of an interface or abstract class.

## 4 Evaluation

The evaluation of our approach consists of several measurements of the operational and reengineering costs for our multi-tenancy support layer. In particular we want to measure the overhead introduced by the multi-tenancy support layer. We compare the results of our multi-tenancy support layer with a multi-instance, single-tenant approach and the default multi-tenant solution without flexibility.

We first describe the general methodology we applied. Next, a general cost model for the operational and reengineering costs of SaaS applications is specified. Finally we present the measurements we performed and compare the results with our cost model.

### 4.1 Methodology

In this evaluation we measure and compare the operational and engineering costs between a default and flexible single-tenant version, a default multi-tenant version (without flexibility), and a multi-tenant version using our multi-tenancy support layer. For these measurements we use the hotel booking application described in the case study. The source code of these four versions including our multi-tenancy support layer, is available on `http://distrinet.cs.kuleuven.be/projects/CUSTOMSS`.

To determine the operational costs the diferent versions of the application are deployed on top of Google App Engine (SDK 1.5.0), using the high replication datastore (default option). In the case of the single-tenant application, we deploy a separate application for each tenant, while both multi-tenant versions only need one application each. Each tenant is represented by 200 users who each execute a booking scenario. This booking scenario consists of 10 requests to the application: first several requests to search for hotels with free rooms in a given period, then creating a tentative booking in one hotel and finally the confirmation of the booking. The different users of one tenant execute the booking scenario sequentially, while the tenants run concurrently. Notice that it is not our goal to create a representative load for this application, but to compare the operational costs of the different versions under the same load. We retrieve the information about the execution cost via the GAE Administration Console. It provides a dashboard displaying the resource usage by the application. The focus of this comparison is on the relative differences between the execution costs, since the absolute numbers depend on the current (global) load on the GAE platform.

The reengineering costs are compared based on the quantity of source code used to develop the case study application for the different versions. We make a distinction between Java code, JSP pages (for the user interface), and configuration files (XML). The number of source lines of code are determined using David A. Wheeler's 'SLOCCount' application.

### 4.2 Cost Model

The goal of the cost model is to define the metrics for our measurements, and to represent our hypothesis about the operational and reengineering costs associated with single-tenant and multi-tenant applications. In addition, it enables us to analyze the impact of customization flexibility on these costs.

**Operational Costs.** The operational cost can be subdivided in (i) the application's execution cost (resource usage), (ii) the costs to maintain the application such as performing upgrades, and (iii) the administration cost, i.e. the cost to provision a new customer (tenant) with an application.
*Execution Cost.* We use CPU time, memory and storage usage as the main execution cost drivers. Another important resource is network bandwidth. However, the introduction of multi-tenancy has no effect on the required bandwidth.

Let $t$ be the number of tenants, $u$ the number of active users per tenant, and $Cpu(t,u)$, $Mem(t,u)$ and $Sto(t,u)$ the total usage of respectively CPU, memory and storage. Then, in the case of a single-tenant application (ST),

$$\begin{aligned} Cpu_{ST}(t,u) &= t * f_{CpuST}(u) \\ Mem_{ST}(t,u) &= t * (M_0 + f_{MemST}(u)) \\ Sto_{ST}(t,u) &= t * (S_0 + f_{StoST}(u)) \end{aligned} \tag{1}$$

where $f_{CpuST}(u)$, $f_{MemST}(u)$ and $f_{StoST}(u)$ are functions of $u$, representing the usage of CPU, memory and storage by one single-tenant application instance. $M_0$ and $S_0$ are constants for the memory and storage usage by an idle instance.

In the multi-tenant case (MT) we introduce an extra parameter $i$, i.e. the number of identical multi-tenant instances managed by a load balancer (see SaaS maturity level 4 in [7]). Then,

$$\begin{aligned} Cpu_{MT}(t,u,i) &= t * (f_{CpuST}(u) + f_{CpuMT}(u)) \\ Mem_{MT}(t,u,i) &= i * M_0 + t * f_{MemST}(u) + f_{MemMT}(t) \\ Sto_{MT}(t,u,i) &= S_0 + t * f_{StoST}(u) + f_{StoMT}(t) \end{aligned} \tag{2}$$

where $f_{CpuMT}(u)$ is a function of $u$, representing the additional CPU necessary for tenant-specific authentication and isolation of the incoming requests. $f_{MemMT}(t)$ and $f_{StoMT}(t)$ are functions of $t$ for the additional memory and storage required to store (global) data about the tenants, for instance the tenant's name and address.

Since the number of multi-tenant instances is limited compared to the number of tenants and the additional amount of memory and storage for multi-tenancy support is relatively small compared to the shared amount of memory and storage ($M_0$ and $S_0$), this results in:

$$\begin{aligned} i &\ll t \\ f_{MemMT}(t) &\ll (t-i) * M_0 \\ f_{StoMT}(t) &\ll t * S_0 \end{aligned} \tag{3}$$

Thus from Equations (1), (2) and (3), we can compare the execution costs of the single-tenant and multi-tenant versions:

$$\begin{aligned} Cpu_{ST}(t,u) &< Cpu_{MT}(t,u,i) \\ Mem_{ST}(t,u) &> Mem_{MT}(t,u,i) \\ Sto_{ST}(t,u) &> Sto_{MT}(t,u,i) \end{aligned} \tag{4}$$

As a result a multi-tenant application consumes less storage and memory than a single-tenant application, but requires more CPU. However, the latter is limited to authenticating the tenant and ensuring isolation.

*Maintenance Cost.* The maintenance cost largely consists of the cost to develop and deploy upgrades to the application. Let $f$ be the upgrade frequency, $i$ the number of instances to upgrade, and $Upg(f,i)$ the total upgrade cost, then:

$$\begin{aligned} Upg_{ST}(f,t) &= f_{DevST}(f) + t * f_{DepST}(f) \\ Upg_{MT}(f,i) &= f_{DevST}(f) + i * f_{DepST}(f) \end{aligned} \tag{5}$$

where $f_{DevST}(f)$ and $f_{DepST}(f)$ are functions of $f$, representing the development and deployment cost of one single-tenant application instance. The number of single-tenant instances equals the number of tenants $t$. Often there is only one multi-tenant application instance that is automatically cloned to spread the load over multiple identical instances, resulting in $i$ being equal to 1. Besides the application, the multi-tenancy support should also be upgraded, but since this is part of the middleware it should not be taken into account here.
*Administration Cost.* For the SaaS provider the administration cost consists of two constant costs: (i) creating and configuring a new application instance ($A_0$), and (ii) provisioning a new tenant with an application ($T_0$), for instance by registering the tenant ID in the application and providing a URL to access the application. Let $t$ be the number of tenants, then:

$$\begin{aligned} Adm_{ST}(t) &= t * (A_0 + T_0) \\ Adm_{MT}(t) &= A_0 + t * T_0 \end{aligned} \tag{6}$$

**Reengineering Costs.** When migrating an application to the cloud, reengineering is required to make use of the available cloud services, for example storage. In addition, making an application multi-tenant results in an additional reengineering cost. The latter is the difference in reengineering costs between a single-tenant and a multi-tenant application, and is dependent on the middleware platform that is used. For example, when an API for multi-tenancy is provided, this reengineering cost stays limited. Without this support, additional development is required to provide tenant-specific authentication and to ensure isolation between the different tenants.

**Impact of Flexibility.** Our multi-tenancy support layer provides multi-tenant SaaS applications with the flexibility to adapt to the different and varying requirements of the tenants. However, this also has an effect on the operational and reengineering costs.
*Operational Costs.* The tenant-specific configuration of single-tenant applications can be set at deployment time. Therefore the effect of tenant-specific variations have a negligible effect on the execution cost of single-tenant applications. Only the base storage $S_0$ will increase with the core application and its features. In the case of the flexible multi-tenant application, CPU usage $f_{CpuMT}(u)$ (see Eq. (2)) will increase because the tenant-specific configuration should be retrieved and activated by the `FeatureInjector`. Further, additional memory ($f_{MemMT}(t)$) and storage ($f_{StoMT}(t)$) is required to store this tenant-specific configuration and the different feature implementations. Though, these differences are not in such quantity that they will affect Eq. (4).

The impact of adding flexibility on the maintenance cost will be especially noticeable in the upgrade frequency $f$, because the features also have to be maintained. Since the tenant-specific configuration of a single-tenant application is set at deployment time, changes to this configuration will require additional work for the SaaS provider ($C_0$). We add an extra parameter $c$, the (average) number of tenant-specific configuration changes which cannot be done by the tenant. Tenants of a multi-tenant application can set their tenant-specific configuration themselves. This results in no maintenance overhead for the SaaS provider.

$$Upg_{ST}(f,t,c) = t * (f_{Upg ST}(f) + c * C_0) \tag{7}$$

For the administration cost, flexibility only affects the initial configuration of the application ($A_0$ in Eq. (6)) for both versions. In the single-tenant case this consists of setting the tenant-specific configuration, while the SaaS provider needs to specify the default configuration for the multi-tenant application.

*Reengineering Costs.* To add the necessary flexibility, multi-tenant applications require development support for the application developers, support to retrieve and activate the tenant-specific configurations when needed, and a configuration interface to let tenants specify their configuration based on the set of available features. Our multi-tenancy support layer provides this support: multi-tenant applications only have to interact with it. This still results in additional but limited reengineering cost, for example to define the variation points, register the features and specify the default configuration. In a single-tenant application additional reengineering is only needed to facilitate the instantiation of a tenant-specific configuration.

Providing tenants with the flexibility to customize the application, also requires the development of the different software variations. However, this is part of the core application development cost and therefore is not taken into account as reengineering cost.

## 4.3 Measurements

We focus on the execution cost of running the different versions on top of Google App Engine, and the reengineering cost. Since the maintenance and administration costs are hard to measure, we refer to our cost model for more details.

*Execution cost.* To determine the execution cost we run the four different versions of our case study application on top of GAE: a single-tenant version, a multi-tenant version, a single-tenant version with variability, and a multi-tenant version using our multi-tenancy support layer. However, we noticed that there is no difference in execution cost between the two single-tenant versions, since all variability is hard-coded. Therefore we only show the results of the default single-tenant version. Furthermore, the storage cost is not measured. Because the case study is not a data intensive application, data usage is too limited to make any conclusions about the storage cost.

In Fig. 5 we present the evolution of the average CPU usage with an increasing number of tenants. The CPU usage by the single-tenant version is linearly proportional to the number of tenants, as in Eq. (1). We also notice that the CPU usage by both multi-tenant versions is also rather linear, but lower than the single-tenant application, which differs with our cost model (see Eq. (2)). However, our cost model represents the usage of CPU by the application, while

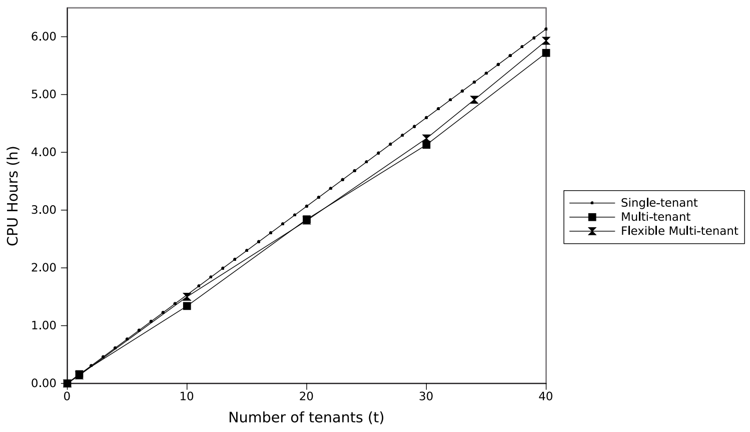

**Fig. 5.** Overview of the CPU usage by the different versions.

on GAE the CPU time for the runtime environment is included. This is an additional cost per application and therefore has more influence on the single-tenant version. We can conclude that the multi-tenant versions require less CPU time than the single-tenant application, and that our multi-tenancy support layer shows limited overhead compared to the default multi-tenant version.

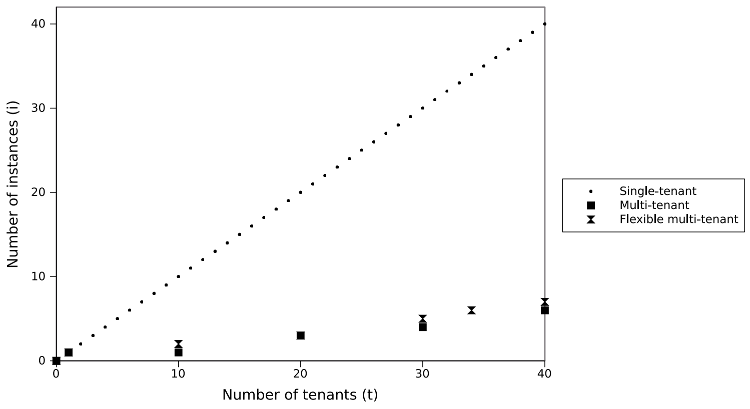

**Fig. 6.** Overview of the number of instances used by the different versions.

The total memory usage cannot be measured precisely, because several other factors despite the application binaries add or reduce memory consumption: a rising number of requests triggers an increase in memory because a new instance (i.e. process required to handle the incoming requests) is started to provide better load balancing, and once the requests decline, instances become idle and are removed to release memory ($M_0$ in Eq. (1) and (2) is 0). Therefore, we use the average number of instances to represent the maximal possible memory usage.

Figure 6 shows the evolution of the average number of application instances when increasing the number of tenants. As can be seen, the difference between the single-tenant and multi-tenant versions is significant. The number of instances for both multi-tenant versions increases only slightly with the number of tenants.

*Reengineering cost.* Table 1 shows the quantity of source code used to develop the case study application. The engineering cost to develop multi-tenancy support is not taken into account, because this is part of the middleware. The differences in lines of source code between the single-tenant and multi-tenant versions is the reengineering cost required to let the application use the multi-tenancy support.

**Table 1.** Overview of the source lines of code (sloc) of the different versions.

| | Java | JSP | XML (config) |
|---|---|---|---|
| Default single-tenant | 915 | 514 | 131 |
| Default multi-tenant | 915 | 514 | 139 |
| Flexible single-tenant | 1016 | 514 | 131 |
| Flexible multi-tenant | 1090 | 514 | 74 |

In the default multi-tenant version without flexibility, the developer only has to write 8 extra lines of configuration compared to the single-tenant version. This is to specify that the `TenantFilter` should be used, which uses the Namespaces API of Google App Engine to ensure data isolation.

When using our multi-tenancy support layer, the difference with the flexible single-tenant application is bigger. However, the majority of these 74 extra lines of Java code are required to use Guice, and not to use our layer. Moreover, the use of Guice resulted in a decrease of configuration lines. Furthermore, in the flexible single-tenant version the configuration is hardcoded and not user friendly. Making this more accessible for the developers to configure will result in more reengineering cost. Finally, we can conclude that adding flexibility to multi-tenant applications by means of our multi-tenancy support layer requires a limited reengineering cost. This cost consists of creating and registering features and their feature implementations, and defining the default configuration.

## 5 Related Work

Related work can be divided into three domains: a) middleware support for developing multi-tenant applications, b) work on customization of multi-tenant SaaS applications, and c) adaptive middleware.

*Middleware Support for Multi-tenancy.* Multi-tenancy is a key enabler to deliver SaaS applications with high cost effectiveness. The current state of the art especially focuses on approaches to support isolation in multi-tenant software applications [15,5]. For instance, Guo et al. [15] discuss design and implementation principles for application-level multi-tenancy, exploring different approaches to achieve better isolation of security, performance, availability and administration among tenants.

Only a few Platform-as-a-Service (PaaS) solutions offer support to build multi-tenant applications. Google App Engine (GAE) [13] facilitates the development of multi-tenant applications via the Namespaces API. Application data is partitioned across tenants by specifying a unique namespace string for each tenant (the tenant ID). These namespaces are supported by several GAE services, such as the datastore and the caching service, enabling tenant data isolation in a transparent way. The Namespaces API is also supported by GAE's open-source implementation AppScale [6]. Other PaaS platforms supporting tenant data isolation are Apprenda SaaSGrid [1] and GigaSpaces SaaS-Enablement platform [12]. None of these platforms directly support tenant-specific customizations and therefore do not offer the same flexibility as our solution. Note that these platforms can also be used as underpinning PaaS for our approach.

In the traditional middleware space JSR 342, the Java EE 7 Specification [9], aims to enhance the suitability of the Java EE platform for cloud environments, including support for multi-tenancy. A descriptor for application metadata will enable developers to describe certain cloud-related characteristics of applications, for example by tagging them as multi-tenant or by specifying the sharing of resources. This extension of the component model with cloud-specific application metadata focuses on persistence and security. Our multi-tenancy support layer, however, offers a way to annotate points of tenant-specific variation, increasing the flexibility of multi-tenant applications, and thus is complementary.

*Customization of Multi-tenant SaaS Applications.* Although tenant-specific customizations are an important requirement [7,30,3], it is not trival to adapt the business logic and data to the requirements of the different tenants [15], especially in Java or .NET, the programming languages commonly used for enterprise applications.

Bezemer et al. [3] applied their multi-tenancy reengineering pattern to enable multi-tenancy in software services. This pattern requires three additional components: a multi-tenant database, tenant-specific authentication and configuration. Configuration is however limited to the look-and-feel and workflows.

In [21] variability modeling techniques from software product line engineering (SPLE) [25] are applied to support the management of variability in service-oriented SaaS applications. Application templates describe the variability via variability descriptors. Our work focuses on the realization of tenant-specific customizations in SaaS applications, which is not covered by this work.

Existing approaches for dynamic customization of multi-tenant SaaS applications utilize dynamic interpreted languages [28,22]. However, we focus on customization of enterprise multi-tier applications, which are commonly written in statically typed languages such as Java or C#. In this context, a dynamic software adaptation approach such as dynamic aspect weaving or dynamic component reconfiguration is preferred.

*Adaptive Middleware.* The state of the art in adaptive middleware [4,18,8,20,26] has mostly focused on adapting applications to one usage context at a time. This means that application software is adapted by replacing an old configuration to a new configuration. In other words, the existing configuration interfaces of adaptive middleware are inherently oriented towards the dimension of the application owner or end user, but have no good ways of managing software variations on behalf of tenants. Adaptive middleware techniques include reflection and aspect-oriented development. The following paragraphs more closely relate our work to these two techniques.

Reflective middleware platforms, such as DynamicTAO [19] and OpenORB [8], provide a configuration interface to inspect and adapt the structure of applications and middleware at runtime. However, these adaptations are based on a global configuration and result in the replacement of components, thus affecting all tenants. They do not allow adaptations scoped to a specific tenant.

Aspect-oriented frameworks such as JAC [24], JBoss AOP [16] and Spring AOP [29], have improved the modularization and customization capabilities of middleware platforms and applications. By means of a declarative configuration application-specific or user-specific extensions can be weaved in where necessary. Currently also dynamic and distributed aspect weaving are supported [24,20,27], including in a reliable and atomic manner [23,32]. These AO-techniques are therefore suitable for usage in a multi-tenant context. Lasagne is an aspect-oriented middleware [33] that supports concurrent, co-existing configurations of the same application instance. This approach is however limited to traditional client-server architectures and does not support customization of multi-tenant software. Still, aspect-oriented software development (AOSD) [10] looks a promising alternative for dependency injection to support tenant-specific injections of crosscutting feature implementations.

## 6 Conclusion

This paper presented a reusable middleware layer on top of an existing PaaS platform to support customizable multi-tenant applications while maintaining the operational cost benefits of true application-level multi-tenancy. We have implemented a prototype on top of Google App Engine and extended the Guice dependency injection framework to achieve activation of software variations on a per tenant basis. This prototype shows improved flexibility with a minimal impact on operational costs for the SaaS provider.

Dependency injection proved to be useful to support the customization of multi-tenant applications. However, adding new features requires the introduction of new variations points in the core application. In addition, for each variation point only one software variation can be injected at a time. This complicates more advanced customizations, such as feature combinations. In this respect, AOSD is a more powerful alternative which we will investigate in the future.

A future research challenge with respect to application-level multi-tenancy is adding support for tenant-specific monitoring and ensuring performance isolation between different tenants. When performing our measurements we experienced that GAE lacks performance isolation between the different tenants. Especially when a number of tenants heavily uses the shared application, this results in a denial of service for the end users of certain tenants. Additional support from the operating system and middleware layers is needed to ensure this performance isolation. Furthermore, tenant-specific monitoring enables SaaS providers to better check and guarantee the necessary SLAs.

**Acknowledgments.** We thank the reviewers for their helpful comments to improve this paper. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

## References

1. Apprenda Inc.: SaaSGrid Middleware. `http://apprenda.com/saasgrid/`
2. Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., Fremantle, P.: Multi-tenant SOA Middleware for Cloud Computing. In: International Conference on Cloud Computing. pp. 458–465. (2010)
3. Bezemer, C.P., Zaidman, A., Platzbeecker, B., Hurkmans, T., Hart, A.: Enabling Multi-Tenancy: An Industrial Experience Report. In: ICSM '10: International Conference on Software Maintenance (2010)
4. Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In: Middleware '98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. pp. 191–206. (1998)
5. Cai, H., Wang, N., Zhou, M.J.: A Transparent Approach of Enabling SaaS Multi-tenancy in the Cloud. In: SERVICES-1 '10: Congress on Services. pp. 40–47 (2010)
6. Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., Wolski, R.: AppScale: Scalable and Open AppEngine Application Development and Deployment. In: International Conference on Cloud Computing. pp. 57–70. (2010)
7. Chong, F., Carraro, G.: Architecture Strategies for Catching the Long Tail. Microsoft Corporation, `http://msdn.microsoft.com/en-us/library/aa479069.aspx` (April 2006)
8. Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N.: The design of a configurable and reconfigurable middleware platform. Distributed Computing 15(2), 109–126 (2002)
9. DeMichiel, L., Shannon, B.: JSR 342: Java$^{TM}$ Platform, Enterprise Edition 7 (Java EE 7) Specification. `http://www.jcp.org/en/jsr/detail?id=342` (2011), Last visited at May 26th 2011
10. Filman, R.E., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2004)
11. Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern. `http://martinfowler.com/articles/injection.html` (January 2004)
12. GigaSpaces Technologies Inc.: SaaS-Enablement Platform for ISVs. `http://www.gigaspaces.com/saas-enablement`
13. Google, Inc.: Google App Engine. `http://code.google.com/appengine/`
14. Google Inc.: Guice. `http://code.google.com/p/google-guice/`
15. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A Framework for Native Multi-Tenancy Application Development and Management. In: CEC/EEE '07: International Conference on E-Commerce Technology and International Conference on Enterprise Computing, E-Commerce, and E-Services. pp. 551–558 (2007)

16. JBoss Community: JBoss AOP. `http://www.jboss.org/jbossaop/`
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. 21, SEI, CMU, Pittsburgh, PA (1990)
18. Kon, F., Costa, F., Blair, G., Campbell, R.H.: The case for reflective middleware. Commun. ACM 45(6), 33–38 (2002)
19. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, C., Campbell, R.H.: Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: Middleware '00: International Conference on Distributed systems platforms. pp. 121–143. (2000)
20. Lagaisse, B., Joosen, W.: True and Transparent Distributed Composition of Aspect-Components. In: Middleware '06: International Conference on Middleware. pp. 41–62. (2006)
21. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. In: PESOS '09: ICSE Workshop on Principles of Engineering Service Oriented Systems. pp. 18–25. (2009)
22. Müller, J., Krüger, J., Enderlein, S., Helmich, M., Zeier, A.: Customizing Enterprise Software as a Service Applications: Back-End Extension in a Multi-tenancy Environment. In: ICEIS '09: International Conference on Enterprise Information Systems. pp. 66–77. (2009)
23. Nicoară, A., Alonso, G.: Dynamic AOP with PROSE. In: ASMEA '05: Workshop on Adaptive and Self-Managing Enterprise Applications. pp. 125–138 (2005)
24. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible solution for aspect-oriented programming in Java. In: REFLECTION '01: International Conference on Meta-level Architectures and Separation of Crosscutting Concerns. pp. 1–24. (2001)
25. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer-Verlag New York Inc (2005)
26. Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., Stav, E.: Composing components and services using a planning-based adaptation middleware. In: Software Composition. pp. 52–67. (2008)
27. Rouvoy, R., Eliassen, F., Beauvois, M.: Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services. In: SAC '09: Symposium on Applied Computing. pp. 1021–1028. (2009)
28. Salesforce.com, Inc.: `http://www.salesforce.com`
29. SpringSource: Aspect Oriented Programming with Spring. `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html`
30. Sun, W., Zhang, X., Guo, C.J., Sun, P., Su, H.: Software as a Service: Configuration and Customization Perspectives. In: SERVICES-2 '08: Congress on Services Part II. pp. 18–25 (2008)
31. Tao, L.: Shifting paradigms with the application service provider model. Computer 34(10), 32–39 (2001)
32. Truyen, E., Janssens, N., Sanen, F., Joosen, W.: Support for Distributed Adaptations in Aspect-Oriented Middleware. In: AOSD '08: International conference on Aspect-oriented software development. pp. 120–131. (2008)
33. Truyen, E., Vanhaute, B., Jørgensen, B.N., Joosen, W., Verbaeten, P.: Dynamic and selective combination of extensions in component-based applications. In: ICSE '01: International Conference on Software Engineering. pp. 233–242. (2001)