Implementation Patterns for Multi-Tenancy

SUMIT KALRA, Indian Institute of Technology Kanpur
PRABHAKAR TV, Indian Institute of Technology Kanpur

Recently multi-tenant applications for SaaS in cloud computing are on rise. These applications increase the degree of resource sharing among tenants with various functional and non-functional requirements. However, often it results in higher design complexity. In this work, we discuss various design patterns to build these applications with efficient tenant management. We divided these patterns in the three categories. The categorization is based on their applicability to design, development, and runtime phases during software development life cycle. These patterns make an application tenant aware and enable multi-tenancy without adding much overhead and complexity in its design.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*(Design) Patterns*

General Terms: Design

Additional Key Words and Phrases: Multi-tenant, Tenant operation and management

ACM Reference Format:

Kalra, S. and Prabhakar, T V. 2017. Implementation Patterns for Multi-Tenancy. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 23 (October 2016), 16 pages.

1. INTRODUCTION

A multi-tenant application can host multiple tenants with heterogeneous requirements on a single instance of the application that provides sufficient degree of variability and customizability [Krebs et al. 2012]. Economical benefits from multi-tenancy results in increased interest in recent years in industry and academic community. A tenant can be either an individual user or an organization (a group of users with same requirements). Another deployment approach to handle tenants with heterogeneous requirements is multi-instance, where a SaaS service provider creates a dedicated instance for each tenant. In a multi-tenant application, a single application instance is shared among multiple tenants. Therefore it leads to increase in the degree of resource sharing and decrease in the operational cost [Warfield 2007]. It also results in lower cost of maintenance in terms of updates and patching as the service provider needs to handle fewer application instances.

The degree of multi-tenancy of an application limits the heterogeneity among tenants' requirements, which can be entertained by the multi-tenant application. Tenants may differ in their functional and non-functional (quality) requirements. Functional requirements include variation in workflow, activation of functionality such as logging and monitoring, analytics etc. Tenants may also differ in their quality requirements such as service level agreements, performance, security, average load.

This work is supported by the MHRD, Govt. of India Grant
Author's address: Sumit Kalra, CS109, Computer Science Department, IIT Kanpur, India; email: sumitk@cse.iitk.ac.in; T V Prabhakar, CS317, Computer Science Department, IIT Kanpur, India; email: tvp@iitk.ac.in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 24th Conference on Pattern Languages of Programs (PLoP). PLoP'17, OCTOBER 22-25, Vancouver, Canada. Copyright 2017 is held by the author(s). HILLSIDE 978-1-941652-06-0

In this paper, we identify implementation patterns for building a multi-tenant application. These patterns help in implementing a multi-tenant application efficiently. Mainly the work focuses on the following questions - 1) how tenants could be defined as a logical entity independent of each other? 2) how much support is required from the underlying system to implement multi-tenancy, 3) how can be application components made tenant aware to generate multi-tenant behavior.

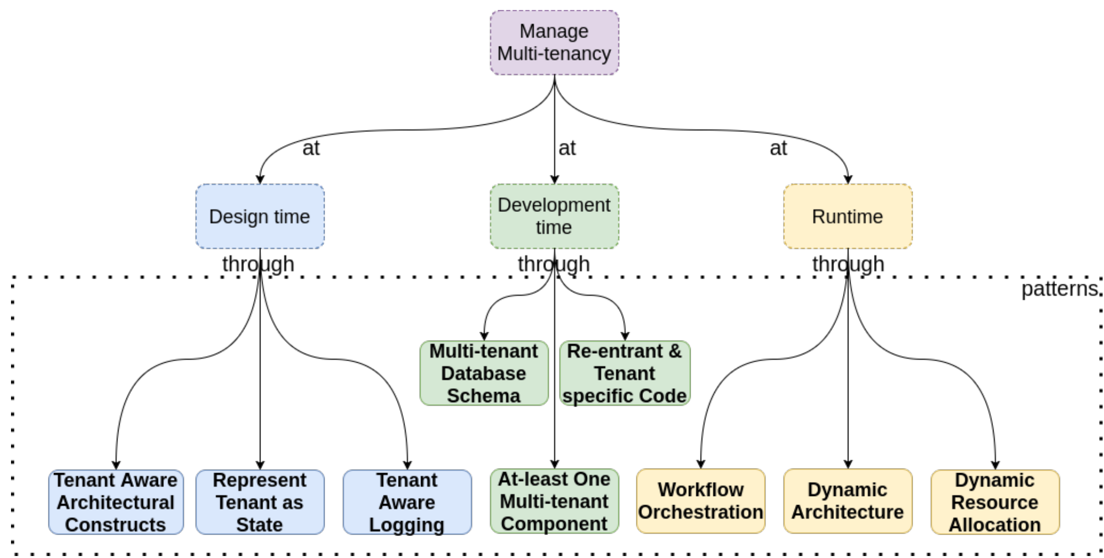

Fig. 1: Multi-tenant implementation patterns and their relationship hierarchy

An overview and relationship among these patterns are shown in the Figure 1. Based on their applicability to design, development, and runtime phases during software development life cycle, we divide these nine patterns in three categories.

—**Design Time:** At the design time, each pattern assumes Tenant as a logical entity. They focus on tenant aware application design that allows taking high-level design decisions to implement, measure and manipulate each tenant independently. Patterns included in this category are:

Tenant as a state

Tenant aware measurement

Tenant aware logging.

—**Development Time:** Patterns in this category work with components of the application to incorporate various tenant management aspects. These patterns require modifying the application components. The patterns in this category are:

At least one multi-tenant component

Multi-tenant database schema

Re-entrant and tenant-specific components

—**Runtime:** The patterns which focus on dynamic binding of components and connectors, are included in this category. They tailor the behavior of the application to meet the specific requirements of tenants at runtime. These patterns are:

Dynamic resource allocation

Dynamic architecture

Workflow orchestration

Although all the patterns require attention during the design phase, this categorization highlights the phases, in which the patterns effectively impact the application behavior. Within a category, patterns are not exclusive of each other. More than one pattern can be applied simultaneously based on the requirements.

1.1 Target Audience

The proposed patterns are useful for architect and software developers of the multi-tenant SaaS applications. These patterns help in the tenant management and handle the multi-tenancy as a primary concern during the design and development phase.

The pattern set proposed here is not exhaustive and requires other patterns to design a cloud-based SaaS application. For example, a multi-tenant SaaS application may use Model-View-Controller design pattern or micro-service based architectural approach along with the patterns described here.

2. REPRESENT TENANTS AS STATE

2.1 Context

In a multi-tenant application, each tenant has its footprint across different components of the application and environment. Tenants differ in their meta-data, configurations and application data from each other. In the multi-tenant scenario, the tenants are logically different entities but share the underlying hardware and software resources.

2.2 Problem

How to embed tenant-specific configurations and meta-data into the application such that it is easy to create, update and delete tenants?

2.3 Forces

- A tenant footprint is spread-over multiple layers of the application such as configuration, database, business logic, and presentation. It is not easy to identify a tenant as a single entity and measure its footprint.
- Tenant may have interleaved footprints. It results in the dependency among tenants.
- If tenants are not logically isolated, *Tenant observability*, *Tenant elasticity* and *Tenant separability* are hard to realize.

2.4 Solution

Encode all the tenant configurations and meta-data corresponding to different layers and app components at one place. These tenant states can be stored in persistent storage such as database or configuration files where it is easy to manage and keep isolated from each other.

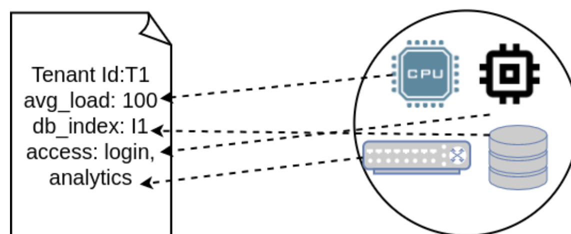

Fig. 2: A sample state of a tenant in the form of configuration file.

Structure: The state of a tenant can be represented as a configuration file as shown in the Figure 2. Other ways to represent tenant states are - a) one configuration file per tenant, b) storing tenant variables in a database.
Dynamics: All the data storage types, used to store tenant state, should be accessible and editable at runtime. A tenant manager extracts tenant-specific information and uses this information to generate tailored responses to the tenant.

2.5 Known Uses

- Most of the implementation methods for building multi-tenant applications boils down to keep the states of two tenant separate from each other. All the tenant properties can be contained in state variables and stored either in persistent storage or application storage [Guo et al. 2007].
- Authors have isolated performance and implemented fairness among tenants of cloud storage using different storage policies related to partition, local weights, replicas and request queue [Shue et al. 2012].

2.6 Consequences

Positive:

- Representing tenants as a state keep all the tenant-specific metadata in one place. It also makes it easy to track a tenant's footprint.
- Tenants' configurations are not mixed with each other. They are independent of each other.
- Since each tenant has its own independent state, therefore it helps in achieving higher degree of *Tenant Observability*, *Tenant Separability* and *Tenant Elasticity*.

Negative:

- Inclusion of a state representation which can be modified at runtime adds performance overhead into the system.
- Sometimes a minor difference in tenants' requirements results in different states and increases the overall number of states to be maintained by the service provider.

3. TENANT-AWARE LOGGING

3.1 Context

A team of software engineers and programmers developing a multi-tenant application. Various patterns and tactics are used to introduce multi-tenant behaviour into the application.

3.2 Problem

Developers and programmers are interested in tracing the activity of each tenant during the development phase. How to segregate systems and application level logs for each tenant?

3.3 Forces

- Since multiple tenants share the application instance. Therefore it becomes harder to trace the individual activity and data flow for each tenant.Tenant information is not present in system level logs.
- A particular component may or may not differentiate among different tenants. In the latter case, all the tenants use the same set of configurations. The metadata collected from these components may not give out tenant-specific details.
- Time series analysis to map system logs to tenants may not be possible due to interleaving nature of tenant tasks.

3.4 Solution

Propagate the information such as tenant id to application logs to map each generated logs to corresponding tenants. The generated log is tenant-aware and can be used to analyze and trace activities of each tenant individually.

Structure: There are multiple ways in which tenant information can be embedded in logs. External components such as load balancer or API gateways can append the log details with tenant information. A tenant itself can also add its information in each of the generated requests. Once the application logs contain tenant information, it is useful for inferring tenant level measurements.

Dynamics: One way to implement this pattern is to append tenant id information with each application request. In this case, the logger module also needs to be modified to handle the appended information. Another workaround is to add tenant id with the existing field in the application logs. It does not require modifying logger module, but log analysis must take this into account to compute tenant level measures.

3.5 Known Uses

—The framework for developing and managing multi-tenant application generates tenant specific logs and metadata information by adding support for tracing each tenant activity [Guo et al. 2007].

—Another example is accountability for data protection in cloud using evidence based event logs [**?**]

3.6 Consequences

Positive:

—The pattern forces to pass tenant information to application logs which can be further mapped to system level logs.

—Even if a component behaves the same with all the tenants, tenant aware logging makes sure that tenant activities are logged, and corresponding tenant level parameters are measurable.

—Tenant aware logging does not require to use time series analysis to segregate logs corresponding to tenants.

Negative:

—Introduces overhead in logging mechanism.

—Requires modifying existing components to generate tenant aware logs.

—It may not be possible for a 3rd party off-the-shelf component to generate tenant aware logs using this approach.

4. TENANT-AWARE ARCHITECTURAL CONSTRUCTS

4.1 Context

Tenant may vary in their non-functional requirements. Some of these quality requirements are agnostic to application logic and solely depends on architectural components of the system.

4.2 Problem

How to measure and control architectural quality attributes such as availability, security, fault-tolerance, recoverability for each tenant independently?

4.3 Forces

—Each tenant is invoiced with the pay-per-use model based on functional and non-functional responses.

—Sharing a single instance makes it difficult to measure and control quality responses for each tenant separately.

—Some of the components are architectural components, which do not depend on the application functionality. These components are deployed only to meet some non-functional requirements. Examples of such architectural components are Load Balancer, Message Queue and State Synchronization amongst replicas.

4.4 Solution

Architectural constructs such as load balancer, responsibility allocations, replications should be configurable to meet the need of tenants. Making the architectural components tenant-aware helps in customizing architectural quality responses for the tenants.

Structure: The configuration of architectural components must be stored for each tenant separately. It can be centrally stored within tenant's metadata for all the architectural components or within each of the components for all the tenants. The configuration parameters depend on requirements of tenants.

Dynamics: Each tenant request is processed using the corresponding policy and configuration settings by the architectural components. The components should be able to load the configuration for each incoming tasks. Along with delivering the responses, it is necessary to generate the separate logs for each tenant to measure the quality attributes at tenant level.

4.5 Known Uses

—In QoS-aware service recommendation, based on tenant's requirement, appropriate service selection is performed. The service selector component is tenant aware but not the functional components [**?**].

—A modeling technique from software product lines (SPL) and model-driven engineering (MDE) can be used to manage dynamic multi-tenancy [**?**].

4.6 Consequences

Positive:

—Tenant aware components allow delivering customized quality responses to the tenants.

—Separate configuration per tenant for architectural components can generate controlled responses and tenant-aware logs. *Tenant-aware Logging* can be used to measure the responses for each tenant individually.

—Tenant-aware architectural components make the multi-tenant application suitable to host the tenants differing in their non-functional requirements.

—If a component is not tenant-aware, a wrapper around a component can be used to generate tenant-aware logs for tenant-level measurements.

Negative:

—Tenants dynamic workload requirements may increase the frequent modifications in configurations of architectural components. The mechanism to update configuration file must be realized optimally to reduce the impact on the performance of the system.

—As architectural components accept configurations at runtime, it increases the complexity of the integration of such components.

5. AT-LEAST ONE MULTI-TENANT COMPONENT

5.1 Context

You have an application with the component-based design. There is a requirement to modify the application to handle multiple tenants with different functional and non-functional requirements. The traditional way of building an application does not address tenants with different requirements. The design of multi-tenant application should be tailored to generate a varying degree of responses to tenants based on their requirements.

5.2 Problem

How to convert an existing application to exhibit multi-tenancy with minimum efforts?

5.3 Forces

—It requires a lot of efforts to identify variation points [Mahdavi-Hezavehi et al. 2013]. They act as control points for varying the degree of responses at runtime.

—It is difficult and may not be desirable, to implement multi-tenancy for each functional and non-functional requirement.

—Implementing multi-tenancy from scratch requires the complex task of redesigning the application and increases the development overhead [Bezemer and Zaidman 2010].

—The multi-tenant SaaS application service provider does not want to increase the operational and development cost.

5.4 Solution

Modify at least one component to deliver different responses that have to be customized for each tenant. In the application stack, even if a single component is tenant aware and behaves differently for each tenant as per their requirements, then it makes the response of whole application multi-tenant to some extent. For example, just making job scheduler in a batch processing application aware of tenants is enough to have some degree of multi-tenancy and vary response time and throughput for tenants.

Structure: As Figure 3 shows a sample workflow of an application. Component A accepts initial input request, and component E generates the final output for the tenants. Every request goes through the component D. It is the only component which is multi-tenant and generates different quality responses for different tenants depending on their requirements.

Dynamics: Depending upon the multi-tenancy requirements, modify at least one component which is present on the critical path of the workflow. Since all the requests of every tenant go through the component on the critical path, therefore the workflow generates the tailored response for each tenant. In the sample Figure 3, a multi-tenant component D makes the response of the whole workflow personalized for tenants.

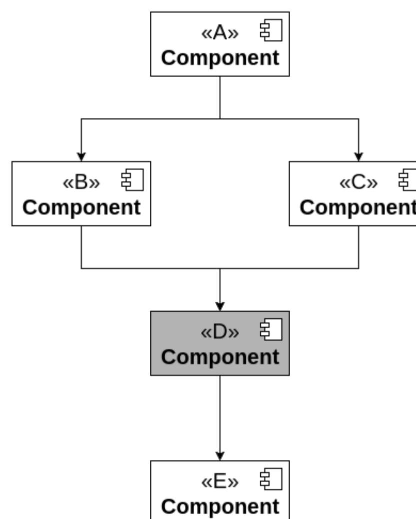

Fig. 3: A single multi tenant component in a workflow can produce tailored responses for different tenants

5.5 Known Uses

—*Google email app for business:* It seems to keep only the user interface themes, logos, etc. different for different organizations and share most of the other application components related to the basic functionality of the email service. Some other multi-tenant components it has for advanced functionality are the ability to share contacts and calendars within the organization [Tuley 2017].

—*Customizable data views:* In this pattern, the data views preferences are stored at client side to display different views to different tenants. It only implements the multi-tenancy at the presentation layer, not at application logic layer [Kabbedijk and Jansen 2011].

—*Module dependent menu:* It provides custom menus to all the tenants containing the link to the only relevant functionality [Kabbedijk and Jansen 2011]. It implements the multi-tenancy by hiding unnecessary functionality to the tenants.

5.6 Consequences

Positive:

—The pattern introduces the multi-tenancy into the application without much redesign tasks related to the identification of variation points.

—It implements multi-tenancy only for the functionality delivered by the selected component(s). It does not implement multi-tenancy for other functionality.

—One can implement multi-tenant application by *Tenant-aware architectural components*, *Dynamic Resource Allocation*, *Multi-tenant Database Schema* and *Dynamic Architecture* also. But all these patterns increase the design complexity of the application. Having a single component in the application which can handle tenants with varying degree of functionality or quality or both, introduces the multi-tenancy into the application to some extent. Since only one component is modified to handle all the tenant related overhead, therefore only design complexity of one component is increased. Rest of the application components have the same design complexity as it was before multi-tenancy.

—Since only a single component is designed to be multi-tenant. Therefore it does not significantly increase the operational and development cost.

Negative:

—The degree of multi-tenancy achieved with having only a single component may be limited. It only provides the multi-tenancy for the functionality and quality related to the multi-tenant component.

—It can only accommodate heterogeneous tenants having variation in the particular requirements which can be met by the multi-tenant component.

6. MULTI-TENANT DATABASE SCHEMA

6.1 Context

The multi-tenant application requires a database to store persistent data for its tenants. Tenants may share the data with other tenants, or all the tenants have their independent data. Since tenants may have slight variations in their functional and non-functional requirements, therefore they may also differ in their data and metadata requirements.

6.2 Problem

How to place a tenant data along with other tenants such that degree of multi-tenancy is high while at the same time, the required degree of isolation for each tenant is maintained and privacy concerns are also taken care?

6.3 Forces

—Different tenants' requirements may map to different incompatible database schemas. The management overhead of handling multiple schemas for different tenants increase the total cost of ownership.

—The heterogeneous schema of tenants may hinder the scalability of the application [Aulbach et al. 2008a].

—Data access pattern for tenants may differ and require different optimization and indexing criteria.

—A tenant may not want to share all the data and schema with other tenants.

—Some of the non-functional responses such as Availability, Persistency have the dependency on the data layer.

6.4 Solution

Design an organic schema for database tables such that single table can accommodate data-specific requirements of tenants. The schema allows sharing the table and data efficiently among tenants having different requirements. Some ways to implement multi-tenant database schema are - separate database, shared database - separate schema, and shared database - shared schema [Microsoft 2017].
Structure: Some of the multi-tenant behavior is implemented at database layer itself. The tenants are not aware of whether the schema belonging to them is part of the multi-tenant schema. The degree of sharing varies among different types of implementation from the separate databases for each tenant to share a single table.
Dynamics: Multi-tenant schema adapt single schema design to handle requirements of multiple tenants. It creates customized indexes and optimized query execution plans to meet the requirements of a particular tenant.

6.5 Known Uses

—*Microsoft Azure SQL database:* It supports multi-tenant applications with the help of multi-tenant data models [Srini Acharya 2017]. It allows designing the multi-tenant database schema to support heterogeneous data requirements of tenants.

—*SQLVM:* It reserves the database resources for each tenant as in case of virtual machines (VM). SQLVM is similar to VM but more lightweight [Narasayya et al. 2013].

—*Chunk Folding:* It is a database schema mapping technique which logically partitions the tables vertically by joining of multiple physical tables [Aulbach et al. 2008b].

6.6 Consequences

Positive:

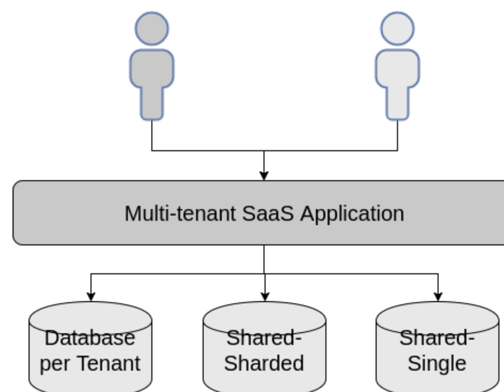

Fig. 4: Different type of data models used to create multi-tenant database schema.

—The multi-tenant database schema makes it easier to handle tenants data. It keeps tenants data logically separate from each other. It maps tenant specific schema on a generic structure.
—It also helps in achieving the higher degree of scalability by optimizing the database resource usage for all the tenants.
—Multi-tenant database schema allows implementing data access policy for tenants to protect data from unauthorized access.
—It is also possible to use different read-write strategies and replicas to deliver variable quality responses such as fault tolerance, recovery, and availability to tenants.

Negative:

—The multi-tenant schema can become complex sometimes if data schemas for different tenants are not compatible.
—Mapping at schema level introduces a time delay in the database query execution.

7. REENTRANT AND TENANT SPECIFIC COMPONENTS

7.1 Context

A new application has to be designed to handle multiple tenants with different requirements. The design of the application should be modular and capable of adding new features on-demand for new tenants.

7.2 Problem

How to distribute or allocate responsibilities among application components to handle a group of tenants?

7.3 Forces

—Tenant may not want to share some of the functionality and data with other tenants due to security and privacy concerns.
—Some of the tasks are common for all the tenants.
—A tenant should be able to maintain its identity while reconfiguring itself or moving between application instances.

7.4 Solution

Analyze the components and categorized them as critical, sharable and optional.

—There are multiple factors responsible for identifying a component as a critical component, and it should not be shared among tenants. Some of these factors are: a) they process confidential data. b) they can't deliver expected quality of service if shared among multiple tenants.
—Sharable components can be shared among all the tenants safely.
—Optional components are not the part of the basic workflow of the application and can be included on demands for the tenants based on their requirements.

Structure: Some components are multi-tenant, and some are not. At the time of provisioning, dedicated components for each tenant is initialized followed by configuring personalized workflow for each tenant using *Workflow Orchestration*.
Dynamics: A tenant's request navigates to the workflow which is designated to the tenant. A privacy check must be included in the critical components of the tenants for preserving the privacy of tenants. The shared components must have sharing policy to share among multiple tenants and satisfy their SLA requirements.

7.5 Known Uses

—A microservice based approach has been proposed to bring the granularity of multi-tenancy at microservice instance level. Tenants can choose which microservices they want to share with other tenants. In this approach, instances of a particular microservice can be running in either shared or non-shared mode [**?**].

—Realizing multi-tenancy using on-demand database service. Rest of the application components are in the reentrant mode, and only database component is tenant-specific component [**?**].

7.6 Consequences

Positive:

—It accommodates tenants having privacy concerns by allowing critical components in non-sharing mode and non-critical components in sharing mode.

—It brings the granularity of multi-tenancy at component level instead of application level.

—Since tenant-specific components are not shared, therefore moving out a tenant from the shared multi-tenant instance is relatively easy.

Negative:

—Overhead of managing different types of components increases the design complexity of the application.

—On-demand inclusion of new components into the workflow results in increased uncertainty in end-to-end SLAs and control the responses for tenants with heterogeneous requirements.

8. DYNAMIC RESOURCE ALLOCATION

8.1 Context

An application executes in a specific environment. The application behavior depends on the application components as well as on the execution environment. A tenant consists of multiple users. The number of users changes over time and results in variable tenant workload. It requires to modify the application behavior for each tenant independently with their dynamic workload and optimize the operational cost.

8.2 Problem

How to deliver services within SLA for co-hosted tenants having variable runtime workload?

8.3 Forces

—The relationship between the application and its execution environment is often complex.

—Tenants are dynamic. Their workload may vary at runtime which may not be predictable and known in advance.

—Communication and dependency among application component can be complex. It may increase the overhead of tailoring the application response for individual tenants.

—The evolving requirements affect addition, modification, and removal of the application component. It makes the application resources allocation and access complex.

8.4 Solution

Control the allocation and access of system and application resources for each tenant at runtime to deliver the desired level of SLA responses. The resources can be software or hardware entity.

Structure: System resources such as processing units, memory, network, and storage are allocated among tenants in either time-shared or space-shared manner. To control these system resources, it requires to profile

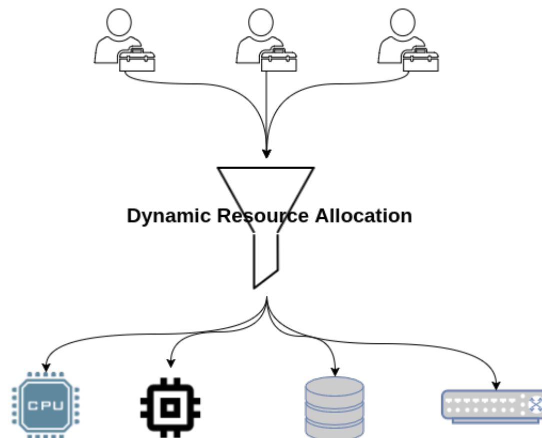

Fig. 5: Dynamic allocation of resources for each tenant at runtime.

the application components in the execution environment and identify the application behavior for given allocated system resources.

On the other hand, to control the application resources, it requires the access of the architectural components of the application. The application-level resources such as component, configurations are shared among tenants using virtual environments.

In some scenarios where the quality of service can be regulated by controlling the allocation of hardware resources, it is also possible to apply this pattern on existing non-multi-tenant applications to generate multi-tenant behavior to some extent.

Dynamics: It needs to identify the application components, respective configurations, parameters and their impacts on the application behavior that helps to map tenant requirements on to the application and environmental parameters. Such parameters are used at runtime to control the behavior of application for each tenant independently.

8.5 Consequences

Positive:

—It allocates resources to tenants dynamically based on their changing requirements.

—Instead of modifying application components and interaction among them, it is easier to change some of the tenant's service quality through controlling the resource sharing and allocation policies.

—Runtime resource allocation accommodates the changing requirements of tenants without modifying the application components.

—The usage of pattern facilitates the inclusion of new component into the application at runtime.

—The process of resource allocation for each tenant request can also be controlled. It allows changing the quality of service for a single tenant based on request type (for example, critical high priority jobs and normal jobs).

Negative:

—It requires complex policy formation to allocate resources optimally at runtime.

—It may not be possible to control desired quality parameters only through controlling the resource allocation. They may require modifications in application components.

—Many times resource usage profile of off-the-shelf components and black-box components is not known. In that case, controlled resource allocation may result in unexpected quality responses for tenants.

8.6 Known Uses

—*Cloud services:* All the cloud service providers control the consumption of resources among tenants to increase the degree of resource utilization at least at the hardware layer. Large-scale data centers also co-locate tenants to use the optimal number of physical machines.

—*Resource allocation in infrastructure networks:* Network infrastructure is shared among a large number of users with different data packets and their priorities. It helps in evolving the resource allocation as per dynamic load on the infrastructure [Kim and Motter 2008].

—*Resource allocation heuristics:* The resource allocation strategy is also useful in improving the energy efficiency of the data center while delivering the desired level of quality of service [Beloglazov et al. 2012].

9. DYNAMIC ARCHITECTURE

9.1 Context

A new tenant comes with some non-functional requirements that are not compatible with group of tenants hosted on existing multi-tenant instance.

9.2 Problem

How to evolve the multi-tenant application to accommodate new incompatible tenant?

9.3 Forces

—Some non-functional requirements can be easily met through modifying the architecture of the application instead of components.

—Since requirements of tenants are not known in advance. Therefore it may not always be possible that initial architectural design of the application can accommodate all the future tenants.

—It is preferable to evolve existing application if possible instead of creating a new one to accommodate new incompatible tenants.

9.4 Solution

Modify the architectural design of the application at runtime. One such approach is to use software connectors among components. Software connectors are useful in changing the architectural design. The new model of the application must be backward compatible to host all the existing and new tenants simultaneously [**?**].
Structure: There are multiple ways to adopt dynamic architecture for a system. Event-based component and connector (C2) style makes the components readily servable from their usage context [**?**].
Dynamics: In event-based C2 style, communication among components are established at runtime based on the tenants' context. The components of the application are loaded for the execution at runtime. It can be useful to add new components or update existing one.

9.5 Known Uses

Aura, an architectural style which supports dynamic mobility of software components based on context and ensures the variable degree of quality-of-service (QoS) [**?**]. Another example of this pattern is mobile middleware platform MobiPads [**?**]. It controls the usage of a resource by middleware and reconfigures it to achieve specific QoS targets. Siena is another example which uses content-based routing to deploy internet-scale publisher-subscriber system [**?**].

9.6 Consequences

Positive:

—Usage of dynamic architecture reduces the need for modifying functional components. It also helps in reducing the testing and debugging efforts.

—It allows to adapt the changes at runtime without requiring the recompile, reload and restart phases.

—It makes the task of accommodating new incompatible tenants relatively inexpensive.

Negative:

—The applicability of dynamic architecture is limited as of now. There have been very few studies on understanding the implications of using dynamic architecture based approaches. The absence of the formal architectural body of knowledge makes harder to realize it.

—It may impact the performance efficiency of the application negatively.

—Sometimes it is not easy to identify which architectural design is suitable for all the new and existing tenants for co-hosting them on a single instance.

—Sometimes migration to new architectural design may not be achievable from the current architectural design of the application.

10. WORKFLOW ORCHESTRATION

10.1 Context

Tenants may have different functional requirements. Some of the functionality may be optional for tenants. The order of execution of application components may also vary from context to context for different tenants.

10.2 Problem

How to facilitate tenants to define their own path of execution based on their requirements?

10.3 Forces

—Highly modular and flexible design may not always be suitable for real-time applications.

—Uses of 3rd party components may not be compatible with the different workflow of native components.

—It is hard to measure the overall quality of service in case of the dynamic composition of services [Liu et al. 2007].

10.4 Solution

Design a multi-tenant application such that the workflow of the application can be orchestrated individually for each tenant to meet their requirements. The workflow orchestration can be either done at provision time or runtime.
Structure: Use of micro-service oriented architecture to design an application is one of the ways to enable workflow orchestration. Implementation of each component as a separate independent micro-service allows the service provider to use on-demand service composition for tenants [Mohabbati et al. 2011]. Tenants can define their workflow. The workflow orchestration can be performed either at the time of provisioning or at the time when a tenant changes its requirements.
Dynamics: Whenever a tenant requirements changes or a new tenant enters into the multi-tenant environment, the multi-tenant system takes the set of requirements from tenants and creates the customized workflow for them. As Figure 6 shows that tenant A and tenant B have different workflows but the same set of components. It also enables tenants to define and modify the workflow using programmable interfaces at runtime.

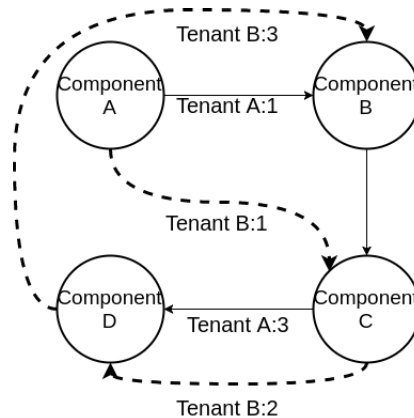

Fig. 6: Two different tenants have two different workflows consisting of same set of components.

10.5 Known Uses

—*IBM Bluemix* The workflow orchestration service is offered by IBM Bluemix [Slominski et al. 2015]. It allows a business user to define workflow based on events, user actions or the context and generate multi-tenant behavior.

—*Workflow Customization* It contains three variations:
a. Exclusive choice: It defines how a single workflow can split into two or more different alternative paths. It is similar to having a different workflow for different tenants. [White 2004]
b. Multiple choices : It also defines multiple workflow paths, but it allows only one path to choose at runtime [White 2004].
c. Deferred choice: It is similar to *Exclusive choice pattern* with one difference - the choice of the path depends on the events during the execution of the process. It tailors the behavior of the application to meet the tenants' requirements [White 2004].

10.6 Consequences

Positive:

—Runtime workflow orchestration allows building a real-time application which can host multiple tenants.

—Workflow orchestration handles the issue of incompatible interfacing among components to some extent.

—Tenant level workflow orchestration compose the services to meet individual tenant's functional and quality of service requirements.

Negative:

—Sometimes workflow orchestration results in non-optimal service composition for a tenant.

—It is not always possible to find off-the-shelf 3rd party components that support dynamic loading and integration into the workflow. It can only use 3rd party components with dynamic loading capability.

11. CONCLUSIONS

These patterns are more significant in a multi-tenant scenario and reduce the design process overhead of the multi-tenant application development. The proposed patterns work along with *The Tenant Manager Pattern* [S. Kalra 2016] and *Patterns for Managing Tenants in a Multi-tenant Application* [S. Kalra 2017] to realize the multi-tenancy. Use of these patterns makes application database, component design and runtime operations tenant-aware and

appropriately generate the responses to satisfy the requirements of respective tenants. All the patterns discussed here focus on the implementation aspect of multi-tenancy. As an extension of this work, we will look for the patterns which can manage multi-tenancy and handle the dynamic nature of the tenants at runtime.

REFERENCES

Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008a. Multi-tenant Databases for Software As a Service: Schema-mapping Techniques. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1195–1206. DOI:http://dx.doi.org/10.1145/1376616.1376736

Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008b. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1195–1206.

Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems* 28, 5 (2012), 755–768.

Cor-Paul Bezemer and Andy Zaidman. 2010. Multi-tenant SaaS applications: maintenance dream or nightmare?. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, 88–92.

Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. 2007. A framework for native multi-tenancy application development and management. In *e-commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*. IEEE, 551–558.

Jaap Kabbedijk and Slinger Jansen. 2011. Variability in multi-tenant environments: Architectural design patterns from industry. *Advances in conceptual modeling. recent developments and new directions* (2011), 151–160.

Dong-Hee Kim and Adilson E Motter. 2008. Resource allocation pattern in infrastructure networks. *Journal of Physics A: Mathematical and Theoretical* 41, 22 (2008), 224019. http://stacks.iop.org/1751-8121/41/i=22/a=224019

Rouven Krebs, Christof Momm, and Samuel Kounev. 2012. Architectural Concerns in Multi-tenant SaaS Applications. *Closer* 12 (2012), 426–431.

Shu-Lei Liu, Yun-Xiang Liu, Fan Zhang, Gui-Fen Tang, and Ning Jing. 2007. Dynamic web services selection algorithm with QoS global optimal in web services composition. *Ruan Jian Xue Bao(Journal of Software)* 18, 3 (2007), 646–656.

Sara Mahdavi-Hezavehi, Matthias Galster, and Paris Avgeriou. 2013. Variability in quality attributes of service-based software systems: A systematic literature review. *Information and Software Technology* 55, 2 (2013), 320–343.

Microsoft Developer Network Microsoft. Retrieved: February 2017. Multi-Tenant Data Architecture. https://msdn.microsoft.com/en-us/library/aa479086.aspx. (Retrieved: February 2017).

Bardia Mohabbati, Dragan Gašević, Marek Hatala, Mohsen Asadi, Ebrahim Bagheri, and Marko Bošković. 2011. A quality aggregation model for service-oriented software product lines based on variability and composition patterns. In *International Conference on Service-Oriented Computing*. Springer, 436–451.

Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. (2013).

P. TV S. Kalra. 2016. The Tenant Manager: A Pattern for Multi-tenant Applications. In *Proceedings of Pattern Languages of Programs*.

P. TV S. Kalra. 2017. Patterns for Managing Tenants in a Multi-tenant Application. In *Proceedings of European Conference on Pattern Languages of Programs*.

David Shue, Michael J Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage.. In *OSDI*, Vol. 12. 349–362.

Aleksander Slominski, Vinod Muthusamy, and Rania Khalaf. 2015. Building a multi-tenant cloud service from legacy code with docker containers. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 394–396.

Steve Stein Debra Dove Srini Acharya, Carl Rabeler. Retrieved: July 2017. Design patterns for multi-tenant SaaS applications and Azure SQL Database. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-design-patterns-multi-tenancy-saas-applications. (Retrieved: July 2017).

John Tuley. Retrieved: July 2017. What is a multi-tenant application? https://quickleft.com/blog/what-is-a-multi-tenant-application/. (Retrieved: July 2017).

Bob Warfield. 2007. Multitenancy Can Have a 16:1 Cost Advantage Over Single-Tenant. *Smooth Span Blog* (2007).

Stephen A White. 2004. Process modeling notations and workflow patterns. *Workflow handbook* 2004 (2004), 265–294.