The Multi-tenant Pattern

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

Traditional multi-user applications are designed to provide the same functional and non-functional responses to all the users. However, customers of the contemporary applications may have different expectations from the application. To design an application with the ability to meet diverse requirements of customers, is a recurring problem in various domains including software applications. This paper presents a pattern called “Multi-tenant” that can be employed to cater such diverse set of requirements. We present a reference architecture of the pattern along with examples and known uses. Our analysis of the various use cases shows that the pattern is useful in designing applications for heterogeneous consumers.

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

General Terms: Design

Additional Key Words and Phrases: Multi-tenant, Reference Architecture, Cloud Computing

---

1. INTRODUCTION

A conventional multi-user application assumes that all the users have the same functional and non-functional requirements. It does not cater to the specific needs of the users. Whereas tenant (consumer) of a contemporary cloud service may differ in functional and non-functional responses such as user interface, resource demand, and response time. A tenant is different from a traditional user as it expects highly customizable services from the application [Bezemer and Zaidman 2010]. It can be an individual or an organization with multiple end users. For example, Dropbox [Drago et al. 2012], which offers cloud storage, is a tenant of Amazon Web Service (AWS) [AWS-EC2 2016]. AWS also has other tenants such as Netflix, Nokia, and Samsung which expect a different set of responses from the service [Services 2016]. This paper presents multi-tenant as a design pattern to develop such applications. In contrast with multi-user applications, a multi-tenant application allows its tenants to have diverse sets of functional and non-functional requirements. Internally, such requirements are handled by a single execution instance. Thus the service provider of a multi-tenant application benefits from the reduced overhead of managing multiple application instances. Therefore, operational cost per tenant is less in multi-tenant application compared to a scenario where each tenant is handled by a dedicated instance [Warfield 2007].

2. EXAMPLE

Massive Open Online Course (MOOC) is an online course which leverages the web to provide education across the globe. Recently there has been quite a surge in MOOCs. A typical MOOC consists of video lectures, discussion forums, assignment and various other activities. Each MOOC would vary in size (number of enrollments), duration of the course, intensity (lectures per week), user interaction, and many such features. MOOC have specific management system which is typically called as MOOC management systems e.g. Open edX [Edx 2016] or mooKit [Prabhakar 2015]. Such MOOC management systems are designed to cater the specific requirements of MOOC and can usually handle multiple MOOCs simultaneously. Each MOOC may have a different emphasis on the services offered by the MOOC management system. In this context, we can term MOOC management system as a case of multi-user application. Here a MOOC would be termed as “Tenant” which has varied requirements.

3. MULTI-TENANT PATTERN

3.1 Context

A service provider offers hosted services such as SaaS, PaaS, and IaaS (Software, Platform, and Infrastructure - as a Service) to its tenants (service consumers). A tenant authorizes its end users to avail the services.

3.2 Problem

When a service hosts heterogeneous tenants, there are the following two primary issues to be handled:

(1) How to satisfy diverse requirements of heterogeneous tenants.
(2) How to optimize the overhead of hosting multiple tenants.

3.3 Solution

Design the application in such a way that it can host tenants with different requirements on a single execution instance. Components of such multi-tenant applications are generic enough to deliver a set of diverse requirements.

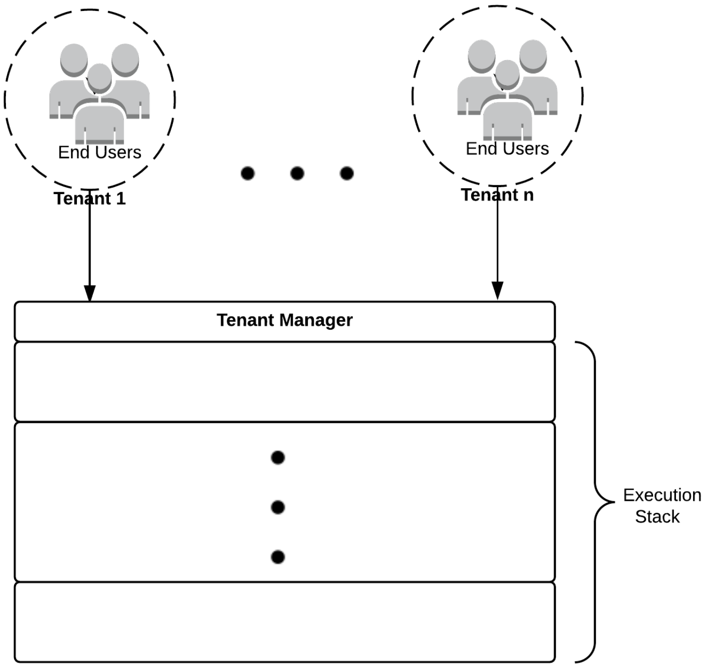
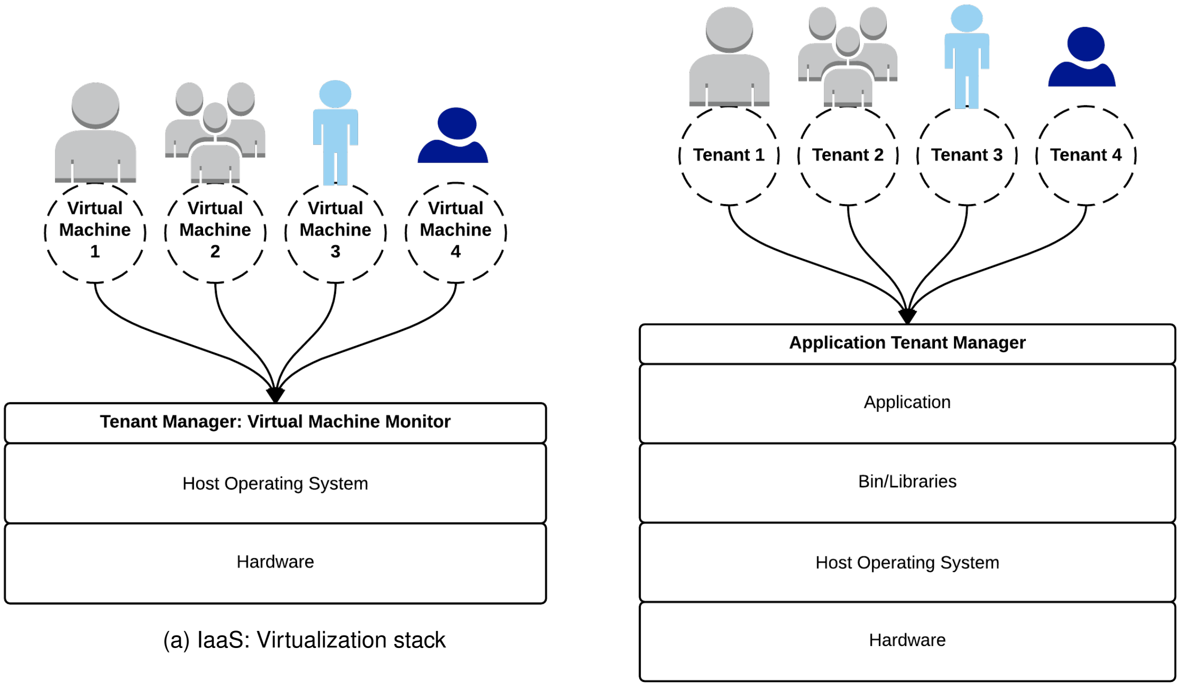


Fig. 1: Reference architecture of the Multi-tenant pattern: Multiple tenants with different requirements shares a single instance of an application.

Structure: Figure 1 depicts the reference architecture for using multi-tenant pattern to share a single instance of the application among multiple tenants. Tenants are oblivious of the fact that application instance is shared with other tenants and deployed on a dedicated server. The reference architecture has following two key conceptual entities: *Tenant Manager* works as a proxy between the application and its tenants, and *Execution Stack* consists of generic components which accept parameters to deliver variations in responses.



(a) IaaS: Virtualization stack

(b) A typical multi-tenant application stack

Fig. 2: Instances of multi-tenant reference architecture

For example, Figure 2 depicts two instances of the reference architecture of the multi-tenant pattern. Figure 2a shows IaaS as an example of the multi-tenant pattern. In this example, hypervisor or Virtual Machine Monitor is the tenant manager which takes care of multiple tenants at hardware and operating system layers. Each VM(tenant) may have a different configuration based on requirements. Another example in Figure 2b shows a typical multi-tenant application stack. Application specific tenant manager handles different tenants and the underlying computation stack is shared via a single execution instance of the application.

Dynamics: Figure 3 shows interaction among the components of the multi-tenant pattern as described below:

(1) A tenant sends a request to the tenant manager to use application functionality.
(2) The tenant manager reads the requirement configuration file.
(3) The tenant manager includes the appropriate parameters corresponding to the requesting tenant.
(4) Modified tenant request is forwarded to the multi-tenant application instance.
(5) The application returns the results to the tenant manager conforms with tenant's functional and non-functional requirements.
(6) Tenant Manager forwards the results to intended tenants.

In this example, the configuration file is used as a repository to store requirement specifications of all the tenants. The application may also directly respond to tenants by-passing the tenant manager in some scenarios.

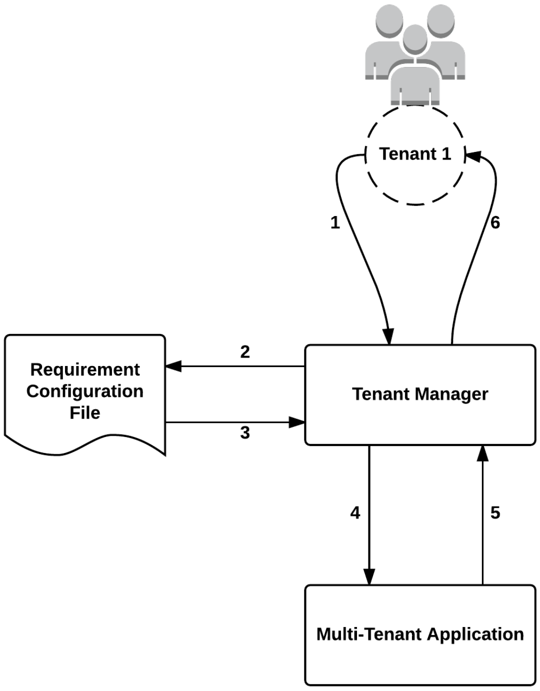


Fig. 3: Interaction among tenants and multi-tenant application components such as Tenant Manager, Tenant QoS configuration and the multi-tenant application

3.4 Implementation

The pattern has two key conceptual entities- *Tenant Manager* and *Execution Stack*. These entities can be mapped to single or multiple modules. There are independent implementation concerns related to each of these entities as follows:

Acquire requirements of the tenants: This concern is related to the tenant manager entity. There are multiple approaches to acquire the different requirements of tenants. Some of them are:

—Tenant mentions the requirements in each request explicitly. These requirements are mapped to parameters of the generic components at runtime.

—Using configuration files or database records, requirements are specified at the time of tenant creation. Tenant manager refers to configuration files for each request.

—Use of a semantic knowledge graph based approach for specifying complex requirements of tenants [Brandt et al. 2008].

Handle tenant requests in compliance with the requirements: This concern impacts the design of generic components of the execution stack. Some of the tactics to design such generic components are:

—Using time-shared resources with different time slices and priorities

—Using software-defined resources such as virtualization, SDN (Software Defined Networking) [Kirkpatrick 2013], SDA (Software Defined Architecture) [Natis 2014] and Network Function Virtualization [Fernandez and Hamid 2015].

4. KNOWN USES

In this section, we will mention some of the known uses from where the pattern has been abstracted out.

- **Salesforce.com:** salesforce.com has improved the SaaS service by implementing multi-tenant applications and bringing down the maintenance cost. They used a meta-data driven architectural approach to design polymorphic applications [Weissman and Bobrowski 2009].
- **Free trials versus premium services:** Various web applications like Github [Dabbish et al. 2012], Dropbox [Drago et al. 2012], Sharelatex [Oswald et al. 2014], and Grammarly [Ubsdell 2012] use a single instance of the application to provide services to the different categories of users such as free users and premium users with different quality. Here, the types of users are tenants in our context.
- **Twitter Alerts:** Twitter has the facility to create alerts and delivers critical information promptly to intended recipients. These alerts are similar to other tweets. Apart from showing alerts differently to the users, alerts are propagated at a much faster rate with time-bound guaranteed delivery. In this case, ordinary tweets and alerts are handled by two different tenants. It makes the Twitter platform a multi-tenant application, which is capable of handling different kinds of tenants (ordinary tweet handler and alert handler) with various functional and non-functional requirements using a single instance of the Twitter application [Twitter 2013].
- **Facebook Promoted Posts:** Facebook has two categories of posts. One is a regular post created by the standard users. Other is a promotional post created by business entities. Promoted posts propagate faster and spread quickly to a larger audience as compared to regular posts. The underlying execution stack is common for both kinds of posts. In this scenario, Facebook is multi-tenant applications whereas regular user and business entities are tenants with different requirements of posts [Inc. 2016].

5. CONSEQUENCES

The pattern has the following advantages:

- **Multi-tenancy as a design concern:** The pattern highlights the issues of handling multiple tenants with different requirements during the design phase. It also makes architects aware of the fact that a multi-tenant concern can be incorporated at various layers such as infrastructure, platform, and software.
- **Higher degree of resource sharing:** The multi-tenant pattern allows co-hosting of tenants with different varying requirements resulting in higher degree of resource sharing as compared to a multi-user scenario. Hence, the operational cost per tenant also goes down.
- **Lower maintenance overhead:** Since the service provider needs to take care of a single application instance, it can bring down the maintenance overhead.
- **Dynamic requirements:** The multi-tenant pattern also facilitates handling of dynamic requirements of the tenants through a tenant manager by implementing it as a runtime entity.

Some of the liabilities of the pattern include::

- **Resource overhead for a multi-tenant pattern:** Use of multi-tenant patterns consumes resources. Although the resource usage overhead can be optimized with efficient implementation, it remains non-zero.
- **Scalability:** Tenant manager becomes a bottleneck if the number of tenants grows. There are implementation variations to handle the scalability issue. Distributed implementation of a tenant manager is possible.
- **Interference:** A tenant can interfere with others if it grows in size or its requirements become more demanding. Tenant Manager keeps track of each tenant and whether their requirements are met or not. It also identifies the tenant who is causing the interference and initiates appropriate actions.
- **Generic application components:** Multi-tenancy requires the design and develop application component to be generic enough to meet variable functional and non-functional requirements. Application components should

also accept parameters related to quality apart from traditional functional parameters. It may result in increased code complexity.

6. EXAMPLE RESOLVED

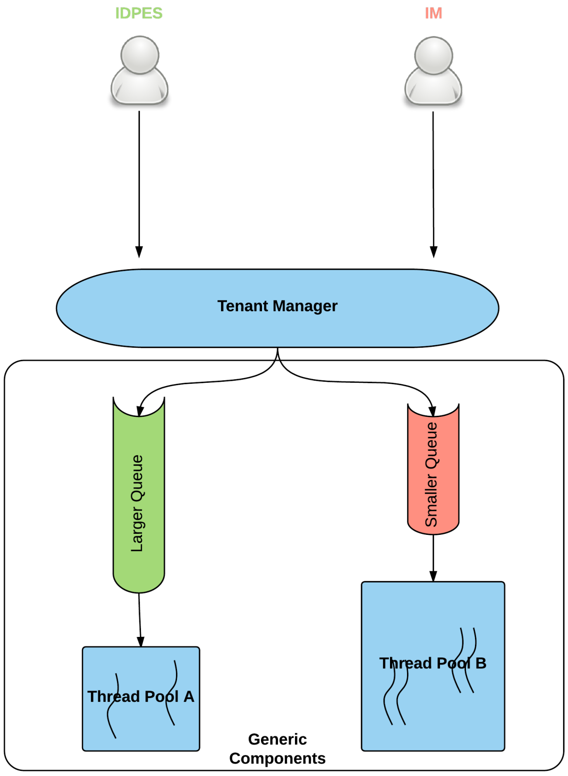


Fig. 4: An example of a multi-tenant message communication system which provides chat and email services

In this section, we show how two widely used systems (email and chat) can be addressed by the proposed multi-tenant pattern. These systems are described as follows.

(1) An intra-domain push email system (IDPES) is used to exchange email only within a domain. It uses push-based approach to deliver a message to the receiver (last mile delivery).

(2) An instant messaging (IM) system is also a message exchange system. IM systems use push technologies to deliver messages.

From a high-level view, IDPES and IM systems have similar functional requirements. However, they differ in non-functional requirements such as the response-time of the message delivery. The IDPES is expected to deliver a message with high response-time. On the contrary, the IM system delivers messages in real-time.

In our example, an organization offers both IDPES and IM services. To reduce the operation overhead it can use the multi-tenant pattern to offer both the services. To realize the same we design a generic application component which uses push technologies to deliver messages. This component can handle the functional requirements of both IDPES and IM systems. However, this application should provide lower response-time to users of the IM system.

In our design, as shown in Figure 4, the tenant manager maintains two thread pools front-ended by queues. Each of the thread pool is responsible for a particular response time. The number of thread in a pool is inversely proportional to its related response-time. Each thread in any pool would represent the generic application component. Upon receiving a request, the tenant manager redirects the request to particular thread pool depending on whether it belongs to the IPDES tenant or the IM tenant. Thus this example shows the applicability of the multi-tenant pattern in such scenario.

7. RELATED PATTERNS

—**Network Function Virtualization:** Network node functions such as load balancer, firewalls are software entities. This design reduces the complexity and increases the ability to control the network functionality dynamically. It is useful in managing the network related services to meet quality requirements of multiple tenants [Fernandez and Hamid 2015].

—**Software Container:** Multiple applications shares a single host operating system, binaries, libraries, etc. and containers allow to run these applications in strongly isolated manner. Considering the applications as tenants and the single host as a multi-tenant application, this pattern is a subset of the multi-tenant pattern [Syed and Fernandez 2015].

8. CONLUSIONS

The multi-tenant pattern brings the issues of handling different requirements of tenants using a shared instance as a primary concern during the design phase. In this paper, we abstract out the multi-tenant pattern from known uses and present a reference architecture for the same. Applications built using this pattern can handle multiple tenants efficiently. The pattern facilitates to host multiple tenants on a single execution instance of the application. At the same time, it also ensures that these co-hosted tenants do not interfere with each other and meet with their functional and non-functional requirements.

REFERENCES

AWS-EC2. Retrieved: August 2016. Elastic Computer Cloud (EC2). `http://aws.amazon.com`. (Retrieved: August 2016).

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.

Sebastian C Brandt, Jan Morbach, Michalis Miatidis, Manfred Theißen, Matthias Jarke, and Wolfgang Marquardt. 2008. An ontology-based approach to knowledge management in design processes. *Computers & Chemical Engineering* 32, 1 (2008), 320–342.

Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 1277–1286.

Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. 2012. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 481–494.

Edx. Retrieved: August 2016. Online courses for free at edX. `http://www.edx.org/how-it-worksRetrieved:2016`. (Retrieved: August 2016).

Eduardo B. Fernandez and Brahim Hamid. 2015. A Pattern for Network Functions Virtualization. In *Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP '15)*. ACM, New York, NY, USA, Article 47, 9 pages. `DOI:http://dx.doi.org/10.1145/2855321.2855369`

Facebook Inc. Retrieved: August 2016. Facebook Business. `https://www.facebook.com/business/`. (Retrieved: August 2016).

Keith Kirkpatrick. 2013. Software-defined networking. *Commun. ACM* 56, 9 (2013), 16–19.

Yefim V. Natis. 2014. Software-Defined Architecture: Application Design for Digital Business. *http://www.gartner.com/webinar/2698619* (2014).

Henry Oswald, James Allen, and Brian Gough. 2014. ShareLaTeX, the Online LaTeX Editor. (2014). `https://www.sharelatex.com/about`

TV Prabhakar. 2015. mooKit Architecture Diagram. (2015).

Amazon Web Services. 2016. All AWS Customer Stories. (2016).

Madiha H Syed and EB Fernandez. 2015. The Software Container pattern. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*.

Bridget Coyne Twitter. 2013. Introducing Twitter Alerts. *Available in: https://blog.twitter.com/2013/introducing-twitter-alerts* (2013).

Gary Ubsdell. 2012. English proofreading software–Grammarly the better choice. (2012).

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

Craig D Weissman and Steve Bobrowski. 2009. The design of the force. com multitenant internet application development platform.. In *SIGMOD Conference*. 889–896.
