akrantha
Thinking action - play it!

Building a SaaS Application

ReddyRaja Annareddy
CTO and Founder

Introduction

As cloud becomes more and more prevalent, many ISV's and enterprise are looking forward to move their services and offerings to cloud. We have been building on premise applications for quite long and have the mastered the art of deploying, managing these applications. Moving to SaaS deployments poses three challenges at the outset, first and foremost security, second scaling and third performance.

As SaaS becomes more prolific, designing, deploying and managing these applications pose a challenge for ISV's and enterprises alike. We discuss here some of the business drivers and considerations in designing a scalable and secure SaaS solutions over time.

Business Drivers

SaaS provides several benefits. The significant being the pay per use model, apart from others. The model is proven and CRM tools like SalesForce, Zoho is an example. The success of these tools has paved the way for the wider acceptance of SaaS services.

SaaS not only reduces the IT foot print in terms of the physical space, servers and other infrastructure, but also eliminates the support, maintenance of the applications.

Last but not the least is the ability of the consumer to choose best breed solutions for the same cost, which means, the organizations no longer have to worry about the innovations and new technology adoptions, which means, that organization can focus on the core business strengths, thus giving them a chance to compete and innovate in their space.

Design Considerations

Multi-tenancy

Multi tenancy is the first and foremost requirement in building a SaaS application. There are several ways to enable the mutitenancy model. The simplest level 1, users the virtualization technology for multi-tenancy. Each tenant gets a virtual machine or a set, which has the complete application. The capacity of the tenant determines the size of the sandbox. This is the case of a same code, same database schema. This model has the advantage, apart from the robust security it offers, has the capability of customizing the software for each tenant and is also a disadvantage, because, this is not a scalable model. The biggest disadvantage is the ability of the SaaS provider to manage the services effectively, even though there are quite number of tools available for deployment and management of virtualization platforms.

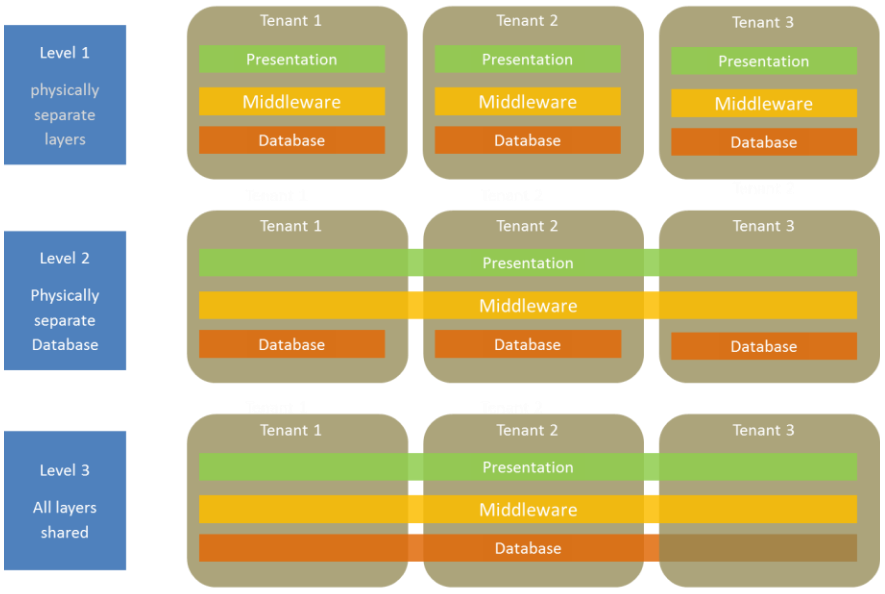


Illustration of Multi-tenancy

The second form or level 2 of Multi-tenancy is to have different database, but the same middleware code. This is generally done by using common cluster of servers for middleware, but an isolated database layer. The advantage with this approach, apart from isolated security, is better manageability of middleware code. The disadvantage is about not able to customize and still have to manage different database instances for data isolation.

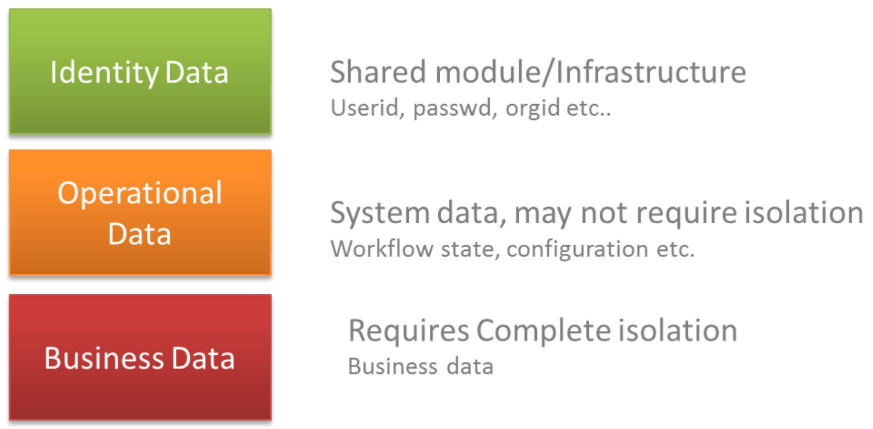
The third or level3 of Multi-tenancy is to share the code and share the same database instance. This has the advantage of being easy to manage, but requires good effort in terms of designing the security model and data isolation. Note that all the three levels offer logical separation.

Security

Enabling more than one customer to use the same system to operate on their precious data is a challenge. Customers have to get the comfort and assurance that, no one else is using the system, and his data is safe and only he can only view the data. This can be addressed easily. However, the challenge comes, when each of the customers has different roles, different permissions to view the data. There could be hierarchy in terms of the application of permission model. In such a scenario, the design of security becomes paramount importance.

Every data object has to be identified with a tenant marker, which is generally the organization id or the tenant id. Secondly, the permissions on this object has to be managed through multiple levels of gated models, so that, enforcement happens at several levels. As the numbers of security gates grow, the performance of the system can become slow.

Data isolation


Identity Data
Shared module/Infrastructure
Userid, passwd, orgid etc..
Operational Data
System data, may not require isolation
Workflow state, configuration etc.
Business Data
Requires Complete isolation
Business data

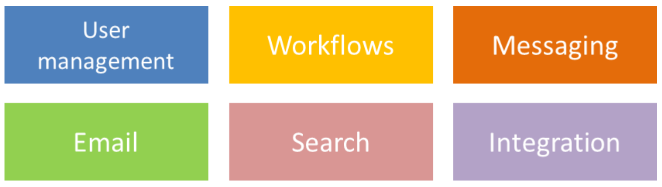
Data types and isolation

There are three types of data generally in the system. The identity data, the operational data and the business data itself. Generally the identity store is common across all tenants in a system. This data is generally of the type userid, password, organization, roles, permissions etc. The operational data is specific to the tenant, but it is not the real data of business value or importance. It is more of workflow state, scheduler information, and customization data and other such data. This data is essential for the operation of the system and has no value from the customer perspective. It is this data, on which the system operates for the function. Some times this is also termed as the meta-data.

Isolation against the business data is the most important function of SaaS system. Generally this is accomplished using separate database schemas or separate servers for database, which allows for daily backup and management of the data. In case of file system data, separate folders, with auto generated ids would be used. The final path should not be exposed to the data. Even better would a document store that does not reveal the type of file system underneath. The document store would isolate users at a higher level, thus filtering the users before accessing the data.

Sharing Common Infrastructure

A significant benefit of SaaS is reducing the total cost of ownership, increasing the value. This is achieved by using the common infrastructure components like Email, Message Bus etc.



Common Services

They perform the service asymptotically and do not worry about tenancy, who is the user etc. These are services which are used across several tenants.

Customization of these services could give some complexity. If at all such requirement exists, it should be done at a different layer, without affecting the shared service.

Scaling, Capacity Allocation and Service Level agreements

Scaling a system should be as simple as adding more number of servers. However scaling can involve scaling the app server and/ or DB servers. In a typical application scaling scenario, application servers are launched or terminated on demand, by simultaneously making changes to the load balancer configuration. Maintaining a single configuration may not be sufficient to get the best out of the hardware. For example, couple of machines in a cluster could have more memory, so that caching sizes could vary. Maintaining multiple versions of these configurations is a challenge. Besides, each of the tenants gets an equal opportunity in using the system.

DB scaling involves many issues. Databases cannot be scaled in the same way as an application server. Some DB scaling techniques involve moving the server to a higher configuration machines, or moving the database to a cluster of machines, where the scaling happens automatically. Scaling the database over several tenants could be a challenge.

SaaS systems face this complexity once the system gets into use and data grows. The performance of the system depends on how well the database has been indexed. At some point the relation databases would pose a severe challenge for scalability. In such cases, NoSQL database should be used. The design should allow swapping of these databases without much change in the overall design and architecture of the systems.

Some of the SaaS systems use both Relational and NoSQL databases. This allows the system to be more transactional and at the same time scalable.

Auditing, Logging and debugging

Auditing involves tracing the actions of a tenant to the lowest level. Auditing multiple log files for customers using the same deployed solution is a challenge. This is part of shared infrastructure. Logging allows the tenant to see only the log in the context of that tenant. Hence tenant id, filter is required or separate logging context for each tenant need to be designed.

Billing, metering and payments

How do we charge the tenants? If it is an hourly basis or user basis, it may not be possible to provide SLA's as one tenant could be pumping lot of load compared to the other. In some cases, the highest paying customer could be denied the performance he deserves. These issues bring the complexity to billing and metering of the systems.

Some system bill based on the number of transactions. In this case metering would track each of the transactions and bill the tenant accordingly. This is the most granular form Billing by tenant user requires some regulation, where the tenant user cannot consume all the resources denying the other tenant, which they rightfully deserve. In such cases, SaaS system requires tenant usage control measures, such throttling etc. Some of the SaaS systems implement throttling and can be configured per tenant.

In some deployments, we have seen customer billing based on the revenue of the company that is handled by that solution. The billing requirements for different customers could be different.

Archiving Data

Shared database poses challenges over isolated database instances per tenant. Database backup and recovery poses a challenge. In case a common database is used, the backup has all the tenants' data. Sometimes it is required to restore only one tenant data. In such a case, the data base recovery procedures could become complex. Customers at some point would want the data locally for audit purposes or for archiving. Some of the SaaS products offer local archiving facility over time, providing the comfort to customers.