PATTERNS OF MULTI-TENANT SAAS APPLICATIONS

Ravi Sharda
Principal Software Engineer
EMC, India Center of Excellence (iCOE)
Ravi.Sharda@emc.com

Manzar Chaudhary
Associate Principal Software Engineer
EMC, iCOE
Manzar.Chaudhary@emc.com

Rajesh Pillai
Senior Consultant
EMC, iCOE
Rajesh.Pillai@emc.com

Srinivasa Gururao
Principal Software Engineer
EMC, iCOE
Srinivasa.Gururao@emc.com

EMC²

Table of Contents

Table of Figures

1. Introduction

Multi-tenancy is a design concept in which a single shared instance of a system serves multiple customers (or even multiple entities/organizations of a single customer), as shown in Figure 1. As opposed to multi-tenant environments, in a single-tenant environment, each tenant has its own application instance, deployed on its independent infrastructure, either on premise or hosted by an Application Service Provider (ASP).

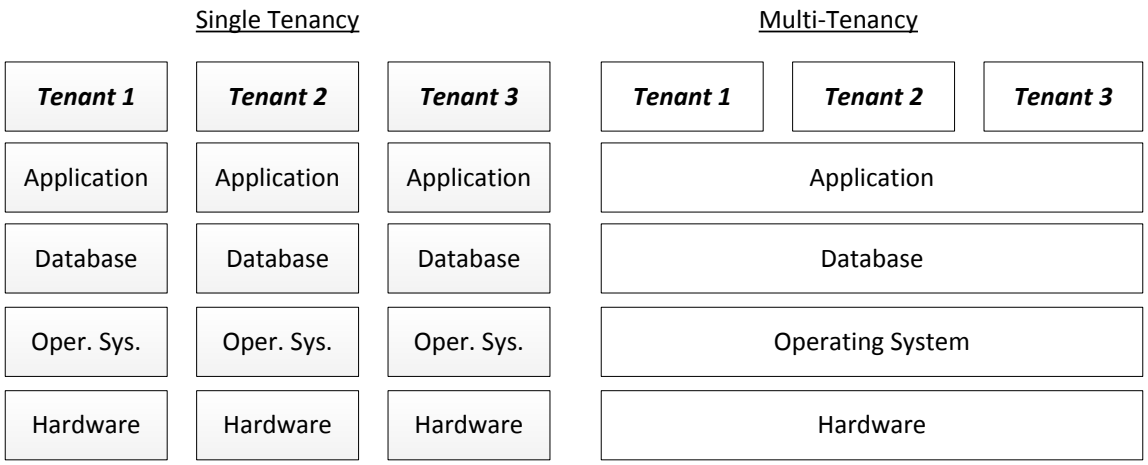


Figure 1: Single- vs. Multi-Tenancy

Software-as-a-Service (SaaS) is a software delivery method through which a hosted software application services the application's functions for multiple customers, eliminating the need for individual customers to deploy and maintain the application on premise. The relationship between multi-tenancy and SaaS is one of an enabler: multi-tenancy enables SaaS.

While the notion of multi-tenancy pre-dates SaaS, techniques for implementing multi-tenancy have become widely discussed only since the advent of SaaS. Despite the plethora of articles available on the Web, many architects and designers find it difficult to gain an understanding of the implementation nuances involved in implementing multi-tenant software. This is where cataloging patterns for implementing multi-tenant SaaS applications would help.

In Section 2 - Architecture Patterns, we cover coarse-grained patterns that address non-local design concerns of a multi-tenant SaaS application – those that apply to most or all of the application. Section 3 - Design Patterns covers patterns that address local design

concerns. Finally, in Section 4 - Implementation Patterns, we present some patterns that are of significance during implementation of a multi-tenant application.

Patterns are organized under the following headings, except when they are grouped together in logical groupings – for example, Design Patterns > Patterns for Authentication.

Heading	Description
Problem	A description of the problem.
Solution	A description of the solution, its applicability, and usage.
Discussion	What pitfalls, implementation nuances, and techniques should one be aware of when using the pattern? How does this pattern relate to other patterns in this document? Are there other names that people use to refer to this pattern?
Consequences	What are the pros and cons of using the pattern?

2. Architecture Patterns

In this section, we cover coarse-grained patterns from the solution domain that address non-local design concerns of multi-tenant SaaS applications. Each sub-section covers a single architectural pattern.

2.1. Application-Instance-Per-Tenant

2.1.1. Problem

You want to host a SaaS application on shared infrastructure, but are not willing to use a shared application. Reasons may be cost of converting an existing application to a shared application, or time-to-market, or isolation.

2.1.2. Solution

Use tenant-specific application instances hosted on shared infrastructure, as depicted in Figure 2.

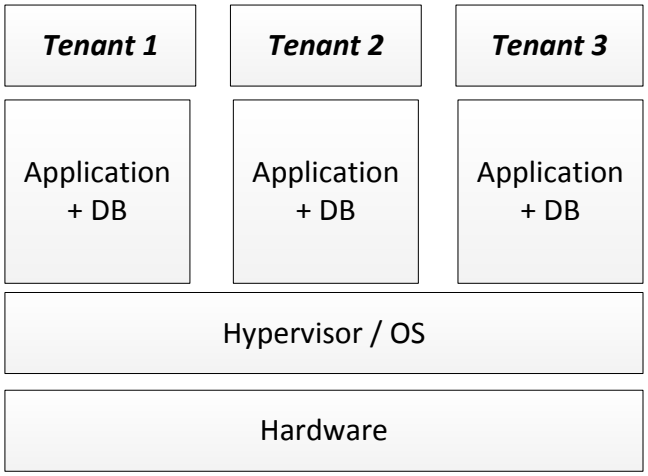


Figure 2: Application-Instance-Per-Tenant on a Shared Hypervisor/OS

A tenant application instance may be hosted as a virtual machine, or on shared hardware/OS, or even on separate physical hosts.

2.1.3. Discussion

Since application instances are provisioned on a per-tenant basis, on-boarding of new tenants/customers can become error-prone and time consuming, especially if installation and configuration steps are complex. Using a virtual machine to host the application instance provides flexibility and other advantages typical to a virtual environment. Using

a template or virtual image as a basis for creating a virtual machine can greatly simplify VM deployment. Hosting application instance on a virtual machine has an additional benefit; it can make it easier to scale the application vertically as capacity demands increase.

Application-instance-per-tenant instantiated as a virtual machine is referred to as “VM-per-tenant”.

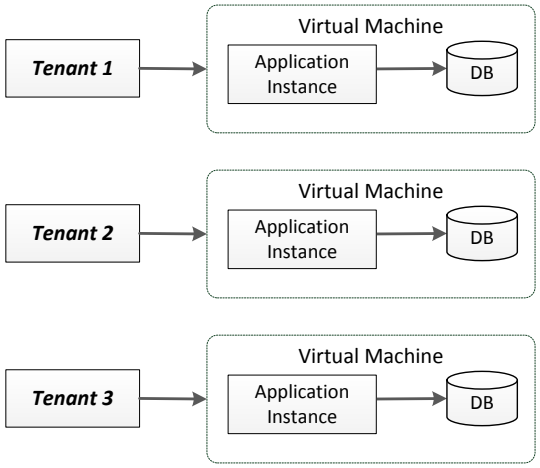


Figure 3: VM-Per-Tenant

Further, a virtual appliance makes provisioning application instances even simpler. A virtual appliance is a pre-packaged software image ready for deployment and running inside a VM and typically consists of a guest operating system and application components. The virtual appliance is typically optimized and pre-configured for the application, so using it to deploy the application usually reduces the steps for application instance provisioning. A virtual appliance may be referred to as a “*Packaged Application Instance*”.

When using a VM-per-tenant as an Application-Instance-per-Tenant, additional security and isolation measures may be necessary. Examples include enclosing a tenant’s instance in a virtual perimeter, hardening the hypervisor and VMs, and so on.

Since per-tenant application instances are separated, while the service is shared (owing to the nature of SaaS applications), the service needs a way to identify the tenant for multiple reasons, one of them being allowing it to route the request to the appropriate application instance. A tenant can be identified through an identifier (Tenant ID) from the

URL or the message header/payload, or through authentication. This is discussed in greater length in Tenant Resolver.

To make this pattern more tangible, Figure 4 illustrates the relationship between the app-instance-per-tenant and a representative set of upstream components.

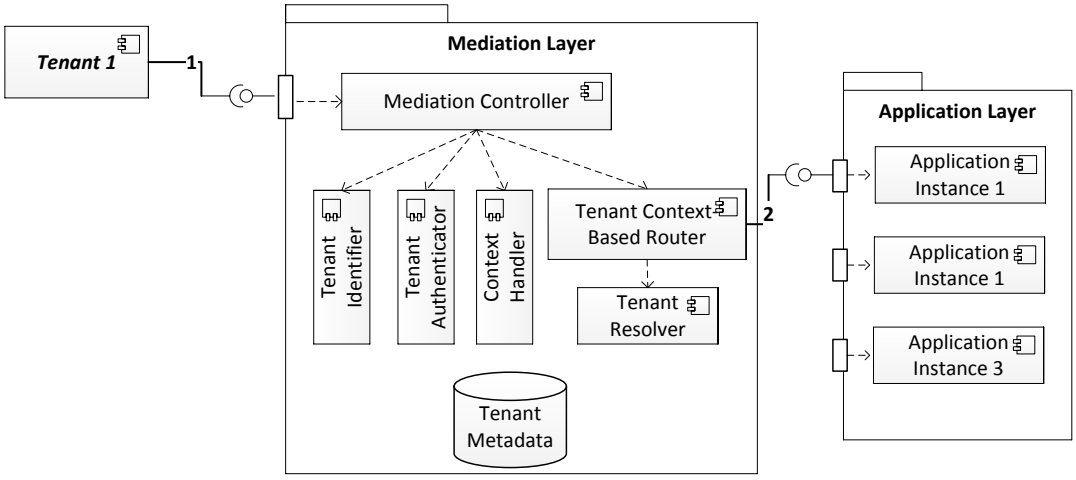


Figure 4: App-Instance-per-Tenant Design

Component	Responsibility
Tenant Identifier	Identifies a tenant (from the URL or payload).
Tenant Authenticator	Authenticates the client.
Tenant Context Handler	Looks up pertinent tenant metadata, binds it to the current thread as Tenant Context, and is passed to Application instances if they need to be tenant-aware.
Tenant Resolver	Determines the application instance to route to, based on the tenant context.
Tenant Context-Based Router	How does this pattern relate to other patterns in this document? Are there other names that people use to refer to this pattern?

2.1.4. Consequences

Among the advantages of this pattern include:

- It is easier to make existing applications multi-tenant using this pattern, since there is little or no change required. This can also reduce time-to-market for companies that want to SaaS-ify their existing products/applications.

- Isolation among tenants is strong, since applications instances for tenants are separated. This addresses concerns such as data leakage, performance, and so on. Similarly, performance or scalability issues can be tackled on a per-tenant basis. Queries of one tenant are less likely to impact others.

- It is easier to perform common maintenance operations on a per-tenant basis. For example, upgrades to the application may be applied to different tenants at different times.

- This pattern makes it easy to customize the application on a per-tenant basis. Customization needs may include tenant-specific field names, user interface look-and-feel, business logic such as conditions for field updates, workflows, etc.

This pattern also suffers some major disadvantages:

- As number of tenants increase, the resource requirements and physical limits of servers and platforms may reduce scalability.

- The overall cost is usually higher, since fewer resources are pooled (for example, per-tenant memory and storage footprint may be high), and there are a number of application instances to maintain. It is hard to leverage economies-of-scale for reducing costs.

- Since upgrades occur on a per-tenant-basis, application upgrade cycles tend to be lengthy and error-prone.

2.2. Shared Application

2.2.1. Problem

You want to host a multi-tenant SaaS application, and leverage economies-of-scale from hosting a common application for a number of tenants.

2.2.2. Solution

Use a shared application to service requests from all tenants. Figure 5 illustrates a shared application.

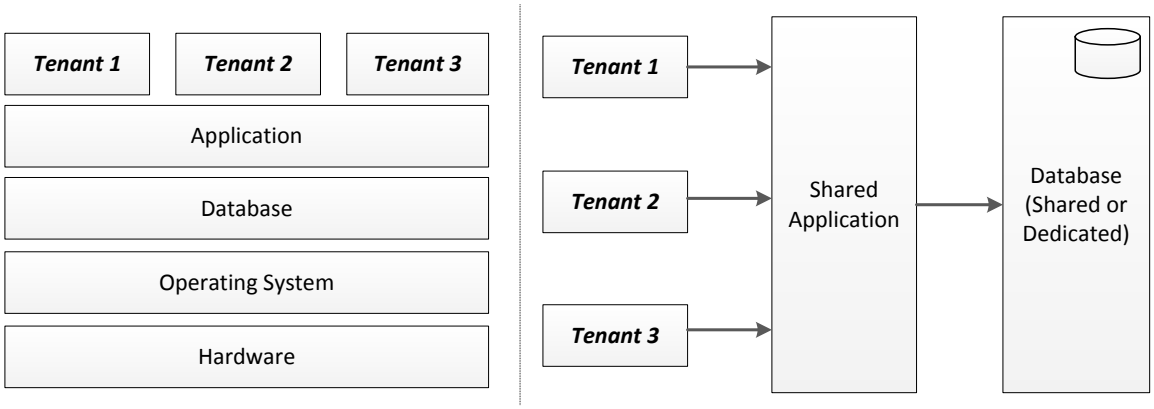


Figure 5: Shared Application

2.2.3. Discussion

When the application is shared across tenants, there must be a way to parameterize the application with a tenant context. This may include the application's interfaces (both API's and GUI's), enabled features based on subscription, database (in case the DB is shared too), configuration files, log files, etc. The basis for parameterization can be a tenant identifier that can take many different forms, some of which explained later.

Although the application itself may be shared, it does not automatically imply that the database is shared as well. All three - Database-per-Tenant, Database Tables-per-Tenant / Shared Schema, and Shared Database Tables – can be used in conjunction with this pattern (Shared Application).

2.2.4. Consequences

Advantages of this pattern include:

- This approach generally reduces cost for the SaaS provider since there is greater consolidation of resources used for servicing many/all tenants,.
- Simplified version upgrades simpler for the provider since all tenants can be migrated to a new version simultaneously.
- In the basic case, the provider needs to maintain a single version of the application, reducing overall maintenance overheads.

Disadvantages of this pattern include:

- Converting an existing on premise application using this pattern takes more work since the application architecture requires change for supporting parameterization.
- Isolation among tenants is much weaker. For example, queries from one tenant may impact others, unless countermeasures for tackling potential resource contention issues are taken.
- Common maintenance operations become more difficult to do on a per-tenant basis. Examples include backups and restores and migration of tenants.
- All tenants must upgrade at the same time, which is not always feasible. This reduces the ability of the SaaS provider to enhance the shared application.
- Customizations on a per-tenant basis are harder to do unless they are carefully designed upfront.

2.3. Database-Per-Tenant

2.3.1. Problem

How to separate databases on a per-tenant basis, for addressing needs such as:

- Data isolation (for example, many companies in the banking and medical domain insist on isolating their data.)

- Flexibility to extend the application's data model to meet individual tenant's needs [Chong]
- Per-tenant restoration from backups
- Avoiding excessive table growth and related management issues, etc.

2.3.2. Solution

Store tenant data in separate databases (DB), regardless of whether the upstream application is shared (Shared Application), or dedicated (Application-Instance-per-Tenant). Databases may be separated using a either a "*Dedicated DB Server*" or a "*Shared DB Server*" as shown in Figure 6.

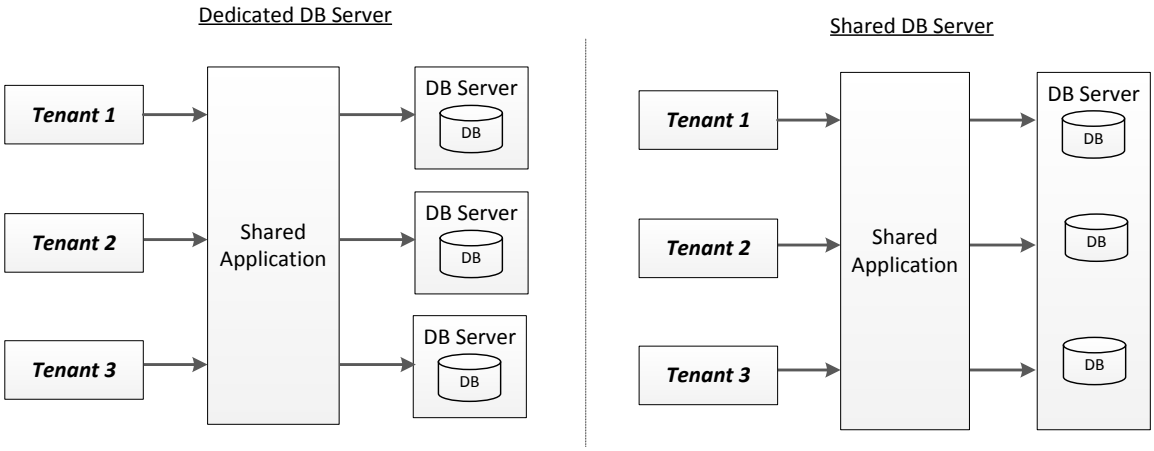


Figure 6: Options for Database-Per-Tenant

2.3.3. Discussion

This approach is generally used in conjunction with shared computing resources and application code [Chong] (Shared Application).

Since each tenant has its own database, a JDBC Connection Pool-per-Tenant is required. When this pattern is used in conjunction with a Shared Application, for each request, the application must identify the tenant and then resolve the right connection pool for DB access. This also means that the application needs to maintain a mapping of which tenant maps to which connection pool.

How do we handle shared data, if tenant databases are separate? One can have a separate database for storing shared data. Whether Database-per-Tenant is used together with Application-Instance-Per-Tenant or Shared Application, a common pool can service all tenants. If there are tenant-specific writes to shared data, the application will need to handle the distinguishing logic.

Variants of this pattern include:

- DBMS process per tenant running on a shared OS
- DBMS process per tenant running on its own VM on a shared hypervisor

Since each tenant has its own database (or even a DBMS or OS), isolation is high among tenants. Access control mechanisms of OS, hypervisor (if applicable), and DBMS can all be leveraged and applied on a per-tenant basis.

This pattern is also referred to as "*Shared Machine*" and "*Separate Database*" by some authors.

Figure 7 illustrates the role of a Database-per-Tenant in a reference architecture of a multi-tenant SaaS application.

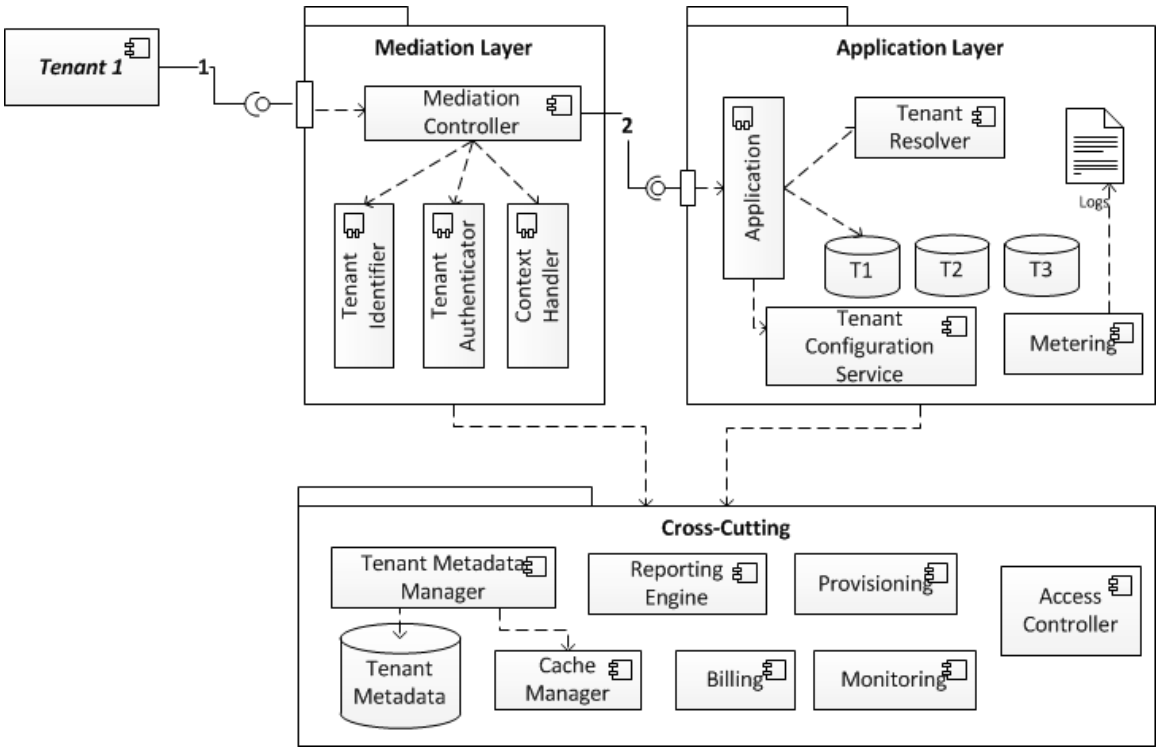


Figure 7: Database-Per-Tenant with Dedicated DB Servers

In the figure above, a request originates from a client of Tenant 1. Within the application layer, the SaaS application ("Application" in the figure) identifies the relevant tenant-specific database (an instance of a database-per-tenant) to connect to using the Tenant Resolver.

2.3.4. Consequences

Benefits of this approach include:

- It is easier to extend application's data model to suit individual tenant's needs. [Chong]
- Restoration of data from backups for recovery purposes tend to become simpler, due to tenant specific database objects.
- Data can be partitioned to achieve scalability goals. For example, one can place different databases into different disks. It also makes it easier to avoid reaching database physical limits due to excessive table growth.

- Allows for different data model versions for different tenants. Further, if a database upgrade requires downtime, only an individual tenant or targeted set of tenants are affected.

Liabilities of this approach include:

- Large number of tenants imply a large number of databases to manage.
- Lack of resource pooling at the database level, leading to proliferation of equipment. Hardware costs tend to be higher than alternative approaches since the number of tenants that can be hosted on a single DBMS server is limited by the number of databases that the server can support [Chong].
- Maintenance is harder, since changes need to be applied in multiple databases.
- Upgrades of database are more complicated: upgrade for each tenant has to be processed individually.
- More storage may need to be allocated per tenant, especially if the database used pre-allocates storage [Stackoverflow]

2.4. Database-Tables-Per-Tenant / Private Schema

2.4.1. Problem

You want to isolate tenant data from that of others, but want to consolidate database infrastructure for tenants.

2.4.2. Solution

Keep tenant's data in tables that are private to an individual tenant. Tenants share a database server, but each tenant has its own private tables (Private-Tables-Per-Tenant with Shared Schema) or its own private schema (Private-Schema-Per-Tenant).

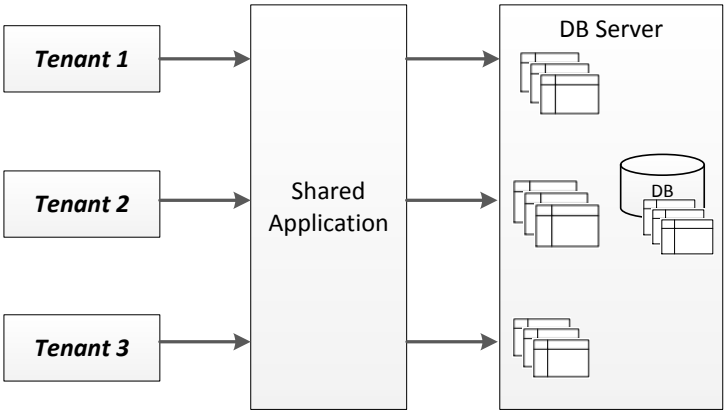


Figure 8: Private-Tables-Per-Tenant

2.4.3. Discussion

One choice to make while implementing this pattern is whether to use tables-per-tenant on a shared schema, or a schema-per-tenant. For most DBMSs there is not much difference with respect to allocated/consumed resources. This is because schemas are usually implemented using a lightweight prefixing mechanism, and does not add much to database overheads.

How do we restrict tenant access to data to the tables they own? Since the database is shared, a database user may have access to all tables, regardless of which tenant the tables belong to.

For better isolation, one may want to place each tenant's data in its own physical table-space. Doing so also makes it easier to migrate tenant's data and to balance I/O workload to different storage backends.

Also, a sort of "table template" can help in provisioning new tenants. Just create tables from the template and modify any tenant-specific metadata.

Another concern that comes up during implementation is how to organize application connection pooling. There are several options:

- *Connection-pool-per-tenant:* Create separate connection pools for the tenants. Select the connection pool, based on Tenant Context associated with the tenant (identified by the Tenant ID).

- Cross-tenant-connection-pool: Use shared connection pool for all tenants. How do we use the shared connections, but still point to tenant-specific schemas/tables? For example, one may use SQL SET SCHEMA upon using a connection to point to tenant-specific schema. One implication of this approach is that the application server must use a common and, potentially, a highly privileged user to connect to the database.

2.4.4. Consequences

Advantages of this pattern include:

- This pattern represents a middle ground between strong data isolation of Database Tables-per-Tenant and consolidation of Shared Database Tables. It supports a moderate degree of data isolation for tenants concerned about security risks of shared tables, while allowing the SaaS provider to leverage common DB resources.

- Makes it easy to extend application's data model to support individual tenant's need, should the need arise.

Disadvantages of this pattern include:

- Can lead to a lot of tables if the number of tenants is high. As number of tables grow, so does maintenance overhead.

- Resource contention is more likely in this case than in Database-per-Tenant. Isolation of tenant's data is weaker that in Database-per-Tenant.

- It is harder to restore tenant data from backups, potentially increasing recovery time. One may have to restore data on a temporary server and import tenant-specific data into production.

2.5. Shared Database Tables

2.5.1. Problem

You want to leverage economies-of-scale by pooling database resources.

2.5.2. Solution & Discussion

This pattern (Shared Database Tables) coupled with Shared Application is what many people think of with respect to multi-tenancy. This topology is illustrated in the Figure 9.

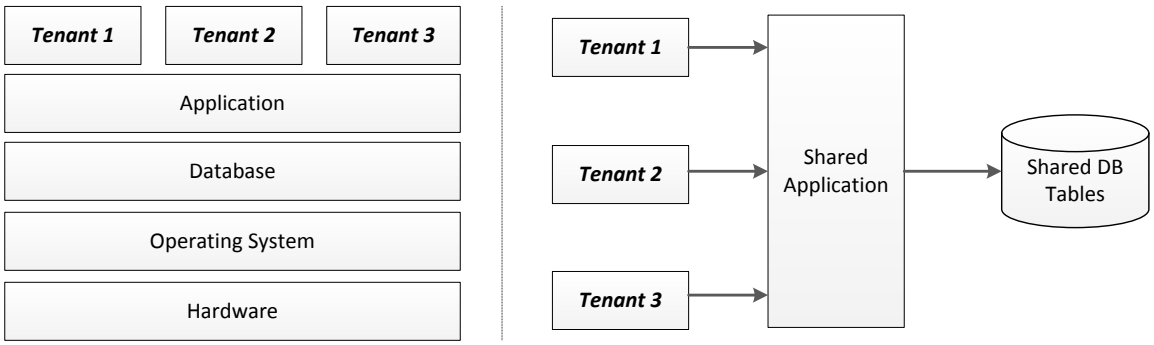


Figure 9: Shared Database Tables Coupled With Shared Application

Since tables are shared across tenants, one needs a "tenant-identifying" (also referred to as "tenant discriminator") column in each database table. At run time, data is filtered based on identified tenant, possibly by adding a default clause such as "*where tenant_id = ?*".

It may make sense to filter data not only for reads, but also for writes – inserts, updates, deletes – for better tenant isolation. Doing so reduces the possibility of one tenant's request inadvertently modifying another tenant's data.

Since the database tables are shared (and by implication, schema and database server instances), connection pool may be shared too. We discuss that in Shared Connection Pool.

Scalability and performance can be improved by partitioning the table using native database partitioning mechanism, by the tenant discriminator column. It enables the database query optimizer to access appropriate partitions given a tenant ID.

When using shared tables for tenants' data, how do you customize tenant-specific data fields? Solutions include Extension Table Layout, Universal Table Layout, among many others.

Several ORM frameworks, APIs, and platforms support multi-tenancy. As an example, Hibernate is called "multi-tenant aware", and supports entity-level discriminator

parameter (as shown in Figure 10) that is passed along to the DB on the JDBC connection for filtering data.

```
@Entity
@TenantDiscriminator(column="tenant_id")
public class SomeEntity {...}
```

Figure 10: Tenant Discriminator in an Hibernate Entity

Java Persistence API (JPA) 2.1, Java EE, and JPA EclipseLink also have support for multi-tenancy.

This approach is especially effective when the tenant-base is high, as benefits of resource pooling can be reaped by the SaaS provider. However, if the tenant-base is very large, the storage space requirements of shared tables may exceed the physical limits of the DBMS server or Storage infrastructure.

2.5.3. Consequences

Advantages of this approach include:

- Better than Database-per-Tenant and Database Tables-per-Tenant / Shared Schema with respect to resource pooling. Since the database tables are same for multiple tenants, connection pools can be shared too.
- Ability to scale up is only limited by the rows and columns the database can hold. In general, scaling up is easier here than in Database Tables-per-Tenant / Shared Schema [Jacobs_Aulbach].
- Administration can be done by executing queries that range over the tenant discriminator column [Jacobs_Aulbach].
- Performing analytics or reporting on customer data is easier.
- All tenants' data is backed up together.

This approach is prone to a number of drawbacks as well, including:

- Since all tenants' data gets backed up together, it is difficult to provide an offsite copy of tenant-specific data to a tenant/customer.

- Restoring an individual tenant's data for recovery purposes is much more complicated. Restoration may require deletion of records for an individual tenant, and reinsertion from a temporary database restored from a backup [Aulbach].

- Isolation among tenants' data is much weaker than Database-per-Tenant and Database Tables-per-Tenant / Shared Schema.

- Performance for one tenant may suffer due to resources consumed by other tenants' queries, administrative tasks such as migration for a tenant, locking conflicts, etc.

- Any schema changes apply to all tenants, regardless of whether they need the change. That may make it difficult to make enhancements to the schema, since changes made for one tenant may not be relevant to another, and therefore may resist any changes in order to avoid disruptions.

- Migration of a tenant requires queries against the operational system [Aulbach].

2.6. Metadata-Driven Architectures

2.6.1. Problem

How do you support extensions to objects in a shared multi-tenant application such that the extensions result in different behaviors for different tenants?

2.6.2. Solution & Discussion

Objects that may require extensions are depicted in color in Figure 11.

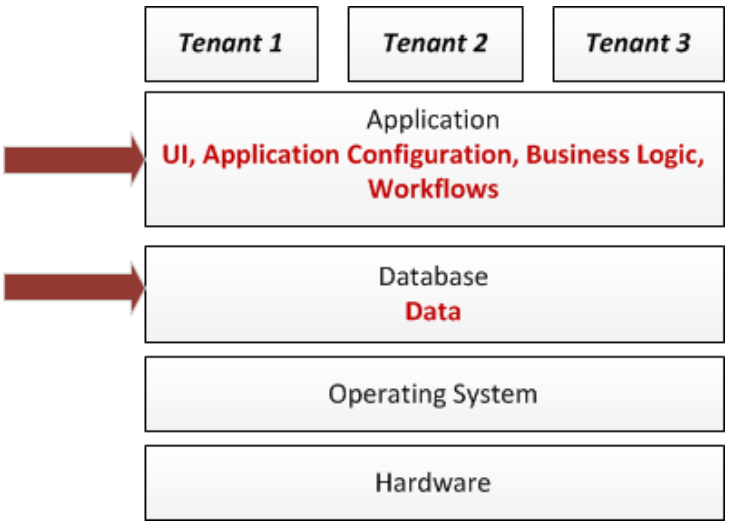


Figure 11: Customizable Objects

They include:

- *GUIs, APIs and forms:* look-and-feel, form/API fields, API versions, etc.
- *Configuration:* policies, encryption keys, PKI certificates, application configuration, user namespaces, backup configurations, and so on.
- *Workflows*
- *Business logic and rules:* e.g. business rules for applying discounts, etc.
- *Data schemas and fields:* e.g. tenant-specific data fields, tables, and even relationships.

A desirable property of customizability in multi-tenant applications is that the behaviors are modified through configuration only – not through application code changes – especially when the tenant-base is growing fast, and many tenants are on-boarded frequently. As Weissman et. al. [Weissman] note, for meeting these challenges "*a multitenant application must be dynamic in nature, or polymorphic, to fulfill the individual expectations of various tenants and their users.*"

To achieve customizability and extensibility on a per-tenant basis the application's objects must be partitioned into base and variable parts, where customizations are required. The base parts instantiate tenant-specific objects at runtime for providing tenant-specific UIs and workflows and using tenant-specific configurations, business logic/rules, and data fields.

Readers may refer to a SalesForce.com paper [SalesForce.com] that provides a detailed description of how Force.com's metada-driven architecture supports customizations/extensions on a per-tenant basis. All major objects – forms, workflows, tenant-specific customizations, etc. – exist only as metadata in a Universal Data Dictionary (UDD), in a few database tables that serve as heap storage [Weissman]. At runtime, "virtual" application components are generated by materializing virtual table data by considering corresponding metadata. Access to metadata by the runtime engine is optimized and the metadata is stored in a cache to improve access performance.

2.6.3. Consequences

This pattern supports tenant-specific customizations/extensions through configuration. Application complexity is high due to generic logic for runtime execution based on metadata, metadata management, etc., and therefore requires careful design. If a SaaS application needs to support only minor customizations for a subset of objects, there are simpler approaches available that employ one or more of these: dependency injection, Aspect-Oriented-Programming (AoP), generative programming, "*Reserved Extended Table Field*" [Sitaram], "Dynamic Extended Sub-Table" [Sitaram], Isolating Filter, GUI templates, "Multi-Entity Support" [Shroff], etc.

3. Design Patterns

In this section, we cover patterns that address local design concerns. Except for the last sub-section, each sub-section covers a single pattern each. Section 3.8 - Patterns for Authentication provides a brief summary of multiple authentication patterns.

3.1. Private Table Layout

3.1.1. Problem

You are using Database Tables-per-Tenant / Shared Schema for servicing multiple tenants. How do you design a table layout that supports tenant-specific customizations?

3.1.2. Solution

Create a copy of a database table for each tenant. Parameterize name of the tenant-specific table with a tenant-specific value, such as tenant ID or name.

An example is shown in Figure 12 (taken from [Aulbach_2]):

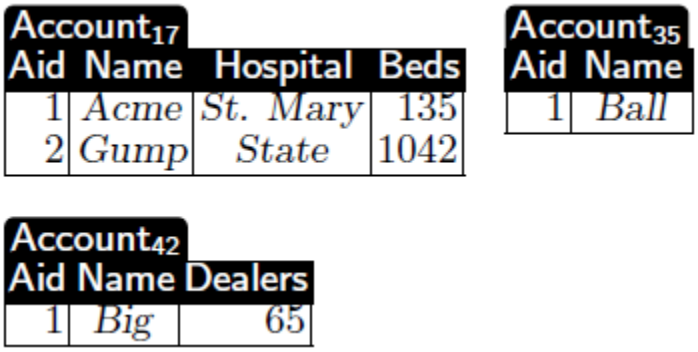


Figure 12: An Example of Private Table Layout (Src: [Aulbach_2])

Figure 12 shows "Account" tables for three different tenants. Notice the difference in field set. This approach allows customization of tables on a per-tenant basis.

3.1.3. Consequences

Key consequences of this pattern include:

- There is no metadata overhead for the application – no additional fields are required in the table for tenant identification.
- This approach allows for tenant-specific customizations, as shown in Figure 9.

- If the number of tenants and number of application tables is high, this layout will quickly lead to a large number of tables.

3.2. Basic Table Layout

3.2.1. Problem

How do you store data for all tenants in shared tables?

3.2.2. Solution & Discussion

The simplest technique for implementing multi-tenancy using Shared Database Tables is to discriminate among tenants' data using a tenant identifying column, such as "Tenant_Id" field in the example shown in Figure 13.

Tenant_Id	First_Name	Last_Name	Age	Deptt	City
1	Foo	Bar	21	Graphics	Chennai
2	Lazy	Guy	28	Reporter	Bangalore
1	Ravi	Sharda	37	IT	New Mars City
2	Manzar	Chaudhary	30	IT	Moon City

Figure 13: Basic Table Layout

Since the data for multiple tenants sits on the same table, discriminator/identifying column must be used with application context to limit what a persistence context can access.

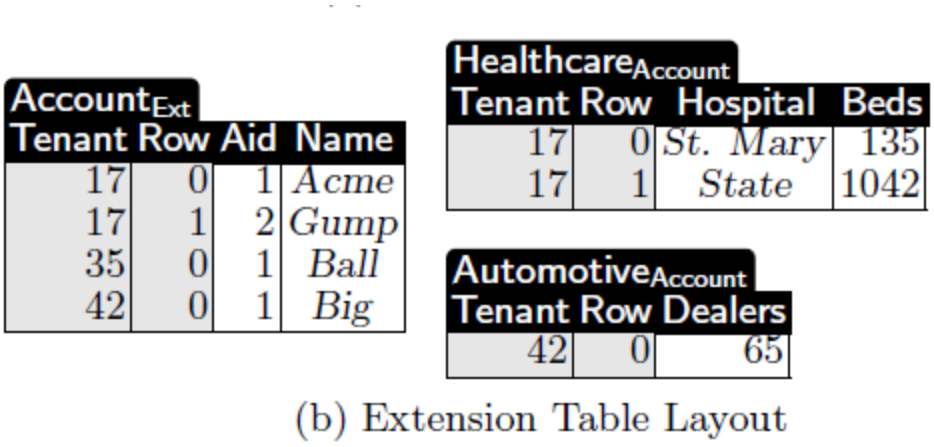
3.3. Extension Table Layout

3.3.1. Problem

How do you support database table extensions for tenants such that multiple tenants share a common extension, and there are multiple such types of extensions.

3.3.2. Solution & Discussion

Among the schema-mapping techniques described by Aulbach et al. in [Aulbach_2], is an "Extension Table Layout", which is a combination of a Basic Table Layout and Private Table Layout. It splits the extensions into separate tables: the base table has columns common to tenants, or the invariant fields. For tenants that require extensions, the extension table holds the extension columns. The extension table is shared by all tenants that require the same extension columns.



$Account_{Ext}$

Tenant	Row	Aid	Name
17	0	1	*Acme*
17	1	2	*Gump*
35	0	1	*Ball*
42	0	1	*Big*

$Healthcare_{Account}$

Tenant	Row	Hospital	Beds
17	0	*St. Mary*	135
17	1	*State*	1042

$Automotive_{Account}$

Tenant	Row	Dealers
42	0	65

(b) Extension Table Layout

Figure 14: An Example of Extension Table Layout (Src: [Aulbach_2])

In Aulbach's example in Figure 11, there is one base table "Account", with fields common to all tenants. There are two extensions of the Account table: Healthcare and Automotive. The extensions are not relevant for tenant with ID 35. Tenant 17 uses extension Healthcare, with fields Hospital and Beds. The gray columns represent the overhead for the meta-data: tenant ID must be stored in both base and extension, since multiple tenants share the tables. Row connects the extensions to the base.

Fetching data at runtime requires additional joins to fetch fields from the extension table. If a query does not reference the extension table, there is no need to read it.

3.3.3. Consequences

Advantages of this pattern include:

- Provides better consolidation than the Private Table Layout. Fewer tables are required for supporting extensions of similar types for multiple tenants.

Disadvantages of this pattern include:

- Application Edit/View master-detail page might be non-performant due to large number of JOINS required to reconstruct logical source table.
- As different types of extensions come up, the number of tables may still experience high growth.

3.4. Universal Table Layout

3.4.1. Problem

You want to use a table structure with immense flexibility.

3.4.2. Solution

Extending the concept of a Basic Table Layout, a Universal Table Layout provides a generic structure, as shown in Figure 15, sourced from [Aulbach].

Universal

Tenant	Table	Col1	Col2	Col3	Col4	Col5	Col6
17	0	1	Acme	St. Mary	135	–	–
17	0	2	Gump	State	1042	–	–
35	1	1	Ball	–	–	–	–
42	2	1	Big	65	–	–	–

Figure 15: An Example of Universal Table Layout (Src: [Aulbach])

As depicted in Figure 15, a Universal Table Layout contains a number of generic fields: Col1, Col2, and so on. These fields are usually made of a flexible type such as “String”, to accommodate string representation of other types such as boolean (trye/false), integer, etc. The “Tenant” field identifies data for a given tenant. The “Table” field refers to the identifier of the table for that tenant.

This layout enables extending the same table in different ways for meeting tenant-specific needs. Since data for all tenants for the table are stored in the same table, one gets consolidation, apart from extensibility.

However, this layout results in a large number of “null” values as all fields/extensions may not be applicable for all tenants. Also, fine-grained support for indexing is not possible.

3.4.3. Consequences

Advantages of this approach include extensibility and consolidation, as explained above.

Disadvantages include:

- This layout may result in a large number of “null” values, and therefore a sparse table, as all fields/extensions may not be applicable for all tenants and all tables.
- Using a generic data type reduces the ability to use DBMS indexing features for fast access.

3.5. Tenant Context

3.5.1. Problem

Multi-tenant applications may sometimes require an object that encapsulates user/tenant information or configuration/system information and references to services, providing such information to other objects through the processing of a service request.

3.5.2. Solution

Use a context object to encapsulate necessary information to be shared throughout the application. A context object can be designed for use through a single request, or a single user session.

3.5.3. Discussion

A context object can be designed for use through a single request, or a single user session.

A tenant context can simply encapsulate a tenant identifier (ID), or may contain additional information such as authentication information, tenant status, reference for tenant configuration, etc.

A tenant context manager object can be made responsible for managing tenant context objects. It creates a new context, returns the context based on an identifier and destroys older contexts.

3.5.4. Consequences

A tenant context encapsulates much context information, relieving other objects from handling non-domain information.

3.6. Tenant Context-Based-Router

3.6.1. Problem

In some multitenant applications, requests need to be routed to the appropriate channel or service so that it can be served.

3.6.2. Solution

Derived from a content-based router introduced by [Hohpe & Woolf], a tenant-context-based-router routes a request to the appropriate service endpoint, based on the Tenant Context. When using Application-Instance-per-Tenant, this routes a request to the appropriate application instance.

3.6.3. Discussion

A router implementation reads the Tenant Context and then routes the request according to the tenant. Some configuration mapping needs to be maintained between the Tenant Context and the route that needs to be taken for the tenant. Whenever a request comes to the router it redirects it to the configured endpoint.

Let's consider an application where we need to have a different request processing workflows for different tenant – meaning each tenant needs to be served differently for the same service call. In this scenario, we can use a tenant context-based routing technique where, on the basis of tenant, we can look up the route path from configuration mapping and then redirect the request to that designated path.

Configuration mapping can have different routing rules. For instance, one set of tenants route to one route while another set can follow a different route. Load balancing techniques can also be used here in order to redirect tenants to different routes based on the load they create. Routing rules can be exhaustive and should be written or configured in a way which can be understood by the Tenant Context-based router.

3.6.4. Consequences

Tenant Context-based routing can be used if we want complete isolation in request processing.

3.7. Tenant Resolver

3.7.1. Problem

A multi-tenant application needs to resolve/identify a tenant. Several processing steps and persistence tasks are required to resolve the tenant.

3.7.2. Solution & Discussion

Tenant resolution is needed in multiple places. Here we consider only the entry point of the SaaS application. Tenants can be identified through query parameters, host headers, authentication, etc.

One way to identify a tenant is through authentication. The application would then be available to all tenants through the same URL, such as http://saas.example.org. The downsides of such an approach are:

- Until the user is authenticated, the application is generic. Only after authentication is a tenant resolved and a tenant-specific user interface can be made available.
- It becomes obvious to the user that the application is multi-tenant.
- This makes it difficult to have more than one identify provider (IdP). One needs to identify the tenant prior to using a tenant-specific IdP for authentication. However, if authentication occurs prior to identification, how do you determine the appropriate IdP? Of course, one way to solve that problem is to ask for a tenant code/name during authentication, apart from the user name. Although, that increases overheads.

Other approaches for tenant resolution are discussed in Sub-Domain-Per-Tenant and Sub-Directory per Tenant.

3.8. Patterns for Authentication

Patterns for application authentication include:

- *Application Managed Authentication:* In this case, a SaaS application maintains a private user directory and manages user accounts [Taneja]. The downsides of

this approach include: a) The responsibility for managing user accounts credentials falls on the SaaS application, forcing it to bear responsibilities that do not lie in its business domain, and b) It forces application users to deal with yet another set of credentials. Additionally, on-boarding new users, migrating users to another provider or on premise, or de-provisioning them becomes more difficult.

- *Directory Synchronization:* In this approach, the application's private user directory is synced with a tenant's user directory [Taneja]. The private user directory is then used by the application to authenticate the tenant's users. Pros and cons of this approach include:
    - Tenant users get to use their existing credentials for accessing the SaaS application.
    - Directory synchronization can easily become a complicated and error-prone process. This makes it difficult for administrators to manage the sync process as the tenant base grows.
- *Customer-Delegated Authentication:* This approach uses a *"token-based authentication and SSO based on directory federation* [Taneja]". In this approach, a local IdP of a tenant issues tokens, such as a Security Assertion Markup Language (SAML) tokens, that are used by the SaaS provider to authenticate users. This approach still requires managing the trust relationship between a tenant and SaaS application and basic user account management, but relieves the provider of managing user passwords and syncing directories [Taneja].

4. Implementation Patterns

In this section, we present some patterns that are significant during implementation of a multi-tenant application. Each sub-section covers a single implementation pattern.

4.1. Isolating Filter

4.1.1. Problem

A multi-tenant application may use a variety of resources such as caches, configuration files, log files, global static variables, etc. How do you limit access of each tenant to only relevant resources?

4.1.2. Solution & Discussion

Use an isolating filter to restrict available resources for each tenant. When available, use built-in filters. Examples of built-in filters include:

- A SQL sub-statement, i.e. “where tenant_id is xxx”
- Tenant-specific log file and format with Log4j Appender
- Tenant-specific prefix/namespaces for business objects in cache
- Tenant-specific files, folders, and files with a prefix
- Tenant-specific XML context/scope in the configuration file

4.2. Sub-Domain-Per-Tenant

4.2.1. Problem

You want to segment/isolate a SaaS application’s interface on a per-tenant basis.

4.2.2. Solution & Discussion

The solution entails segmenting the application into separate sub-domains - one for each tenant. For example:

```
http://tenant1.example.org
http://tenant2.example.org
```

If the application communicates with clients using secure communication protocol Hypertext Transfer Protocol Secure (HTTPS), a separate per-tenant certificate may need to be installed on the server hosting the application's interface, each identifying a single sub-domain (such as tenant1.example.org) within a given domain (example.org). Alternatively, one may install a wildcard SSL certificate that match all of a domain's single-level sub-domains.

Using this approach, an application can do first-level identification of the tenant. If a request comes on tenant1.example.org for example, you know that the IdP for authentication should correspond to tenant1's. Similarly, a proxy can route the requests to appropriate end-points based on the subdomain itself.

4.2.3. Consequences

Consequences of this pattern include:

- This pattern makes it easy to identify a tenant.
- User interfaces can be customized on a per-tenant basis.
- While enabling SSL, buying a wildcard certificate for multiple subdomains is usually relatively expensive.

4.3. Sub-Directory per Tenant

4.3.1. Problem

You need to isolate entry points for tenants, but do not want to use Sub-Domain-Per-Tenant due to some constrains; i.e. the high cost involved to procure a wildcard SSL certificate.

4.3.2. Solution & Discussion

Segment the application by a tenant-specific sub-directory. You have the same domain for all tenants, with a tenant-specific sub-directory. A variant is when tenant ID is passed as a query parameter.

```
-- Tenant Id in the URL
https://example.org/tenant1
https://example.org/tenant2
```

```
-- Variant: Tenant ID in the query parameter
https://example.org?c=tenant1
```

The application itself needs to have separate directories for each tenant. All tenant-related content needs to be available in these tenant-specific directories. Sometimes, there may be redundant content.

First-level tenant resolution is required as the request itself points to tenant directory.

4.3.3. Consequences

Consequences of this pattern include:

- First-level tenant resolution is done, so using tenant-specific UIs and IdPs is possible.
- A single SSL certificate identifying a single domain is used for all tenants, since the same domain services all tenants.
- It becomes obvious to the user that the application is multi-tenant.
- Sub-Domain-Per-Tenant is considered more secure when compared to this pattern (Sub-Directory per Tenant). For example, certain XSS vulnerabilities cannot affect sub domain model because of the same origin policy. Further, this pattern is more prone to Predictable Resource Location attack.

4.4. Connection-Pool-Per-Tenant

4.4.1. Problem

How do you separate database connections used by for different tenants?

4.4.2. Solution & Discussion

Maintain a separate connection pool for each tenant. When using Database-Per-Tenant, any connection pooling will be per-tenant [Jboss]. Define a per-tenant connection pool and select the right pool based on the tenant associated with the user (end user or client application) sending the request.

When using a private-schema-per-tenant (see Private Schema), connection-pooling would again be on a per-tenant basis. If the driver or the connection pooling mechanism supports naming a schema for use for its connections, one could specify one schema per connection pool to achieve Connection-Pool-Per-Tenant [Jboss].

4.4.3. Consequences

Having different connection pool provides control and flexibility in terms of how the tenant app connects to database. Using connection pool per tenant has a lot of advantages. If the application permits, we can have a choice of databases for the tenant. Database-specific resources can be managed at tenant level. The number of database connections/queries at tenant level can be controlled if quota on database resources per tenant is applicable. Due to different connection pool, more stringent security measures can be implemented as tenants have access to only their own connection pool. Tenant-specific maintenance in this model is easier, as tenant-specific database instances can be upgraded without affecting other tenant database instance.

4.5. Shared Connection Pool

4.5.1. Problem

How do you pool connection resources for all tenants?

4.5.2. Solution & Discussion

Use a shared connection pool for servicing all tenants.

When using a private-schema-per-tenant, tenants need access to separate schemas. One way to achieve that is through a shared connection pool. Connections can point to the database itself, using a default schema. However, at runtime, connections can be altered using the SQL SET SCHEMA command (or its variant) depending on which tenant the request is for. This approach provides a single connection pool for many tenants.

4.5.3. Consequences

Consequences of this pattern include:

- Coarse-grained (and possibly a privileged admin user) DB user credentials must be used, so that it has access to collocated tenants' data.

- Administration overhead is lower since connection pool is shared.
- Connection pools can be consolidated to a fewer number.
- Not all databases support SQL SET SCHEMA.

5. Conclusion

Multi-tenant design enables SaaS and enables a SaaS provider to leverage economies of scale by sharing resources across tenants. A range of options are available for implementing multi-tenant software with varying degrees of resource pooling and sharing. In this article, we introduced some known solutions for recurring problems and issues that one may run into when using the solution.

The list of patterns we discussed is neither comprehensive nor exhaustive. The patterns we selected for this article were among the most commonly used for multi-tenant software, from our experience. Among those omitted are Chong's [Chong] "*Trusted Database Connection*", "*Tenant Data Encryption* ", and "*Secure Database Tables*" and Aulbach's [Aulbach_2] *"Pivot Table Layout*", "*Chunk Table Layout*", and "*Chunk folding*".

6. References

[Chong] Chong et. Al., "Multi-Tenant Data Architecture", MSDN, 2006

[Stackoverflow] http://stackoverflow.com/questions/12056145/virtual-segregation-of-data-in-multi-tenant-mysql-database?rq=1

[Aulbach] Stefan Aulbach, Schema Flexibility and Data Sharing in Multi-Tenant Databases

[Aulbach_2] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi Tenant Databases for Software as a Service: Schema-Mapping Techniques. In Wang (2008), pages 1195-1206. ISBN 978-1-60558-102-6.

[Hohpe & Woolf] Hohpe, Gregor and Woolf, Bobby, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison Wesley (2003), ISBN 0-321-20068-3

[Stackoverflow_2] http://stackoverflow.com/questions/8340721/multiple-schemas-versus-enormous-tables/8343142#8343142

[Jacobs_Aulbach] Dean Jacobs and Stefan Aulbach, “Ruminations on Multi-Tenant Databases”

http://d-nb.info/1019589965/34

[Schaffner] Schaffner, Jan, "Multi-Tenancy for Cloud Based In-Memory Column Databases", Srpinger, ISBN: 978-3-319-00496-9

[Weissman] Craig D Weissman et. al, The Design of the Force.com Multitenant Internet Application Development Platform, http://cloud.pubs.dbs.uni-leipzig.de/sites/cloud.pubs.dbs.uni-

leipzig.de/files/p889-weissman-1.pdf

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA. Copyright 2009 ACM

[Jboss] http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html/ch16.html

[Shroff] Shroff, Gautam. Enterprise Cloud Computing: Technology, Architecture, Applications. Cambridge University Press. (2010)

[Mietzner] Mietzner, et. Al, "Horizontal and vertical combination of multi-tenancy patterns in service oriented applications", Enterprise Information Systems

[Sitaram] Sitaram et. al., Moving To The Cloud: Developing Apps in the New World of Cloud Computing. Syngress Publishing. (2012)

[Chong_2] Frederick Chong and Gianpaolo Carraro, "Architecture Strategies for Catching the Long Tail" (2006), http://msdn.microsoft.com/en-us/library/aa479069.aspx

[Gao] Bo Gao et. al., "Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware: Part 4: Design patterns for sharing resources in single instance multi-tenant applications", http://www.ibm.com/developerworks/webservices/library/ws-multitenantpart4/

[Krebs] Rouven Krebs, "Architectural Concerns in Multi-Tenant SaaS Applications"

[Poddar] Indrajit Poddar, "Develop and Deploy Multi-Tenant Web-delivered Solutions Using IBM Middleware: Part 5: A mediation approach for multi-tenancy and three implementation options",

http://www.ibm.com/developerworks/webservices/library/ws-multitenantpart5/

[Taneja] Deepak Taneja, “Identity And Access Management From The SaaS Application Perspective”, CloudTweaks, retrieved from http://www.cloudtweaks.com/2013/05/identity-access-management-perspective-saas-application on 15-Jan-2014

[SalesForce.com] The Force.com Multitenant Architecture: Understanding the Design of SalesForce.com’s Internet Application Development Platform, SalesForce.com