Multi-tenant SaaS database tenancy patterns

04/01/2018 12 minutes to read Contributors

In this article

When designing a multi-tenant SaaS application, you must carefully choose the tenancy model that best fits the needs of your application. A tenancy model determines how each tenant's data is mapped to storage. Your choice of tenancy model impacts application design and management. Switching to a different model later is sometimes costly.

This article describes alternative tenancy models.

A. SaaS concepts and terminology

In the Software as a Service (SaaS) model, your company does not sell *licenses* to your software. Instead, each customer makes rent payments to your company, making each customer a *tenant* of your company.

In return for paying rent, each tenant receives access to your SaaS application components, and has its data stored in the SaaS system.

The term *tenancy model* refers to how tenants' stored data is organized:

- *Single-tenancy:* Each database stores data from only one tenant.
- *Multi-tenancy:* Each database stores data from multiple separate tenants (with mechanisms to protect data privacy).
- Hybrid tenancy models are also available.

B. How to choose the appropriate tenancy model

In general, the tenancy model does not impact the function of an application, but it likely impacts other aspects of the overall solution. The following criteria are used to assess each of the models:

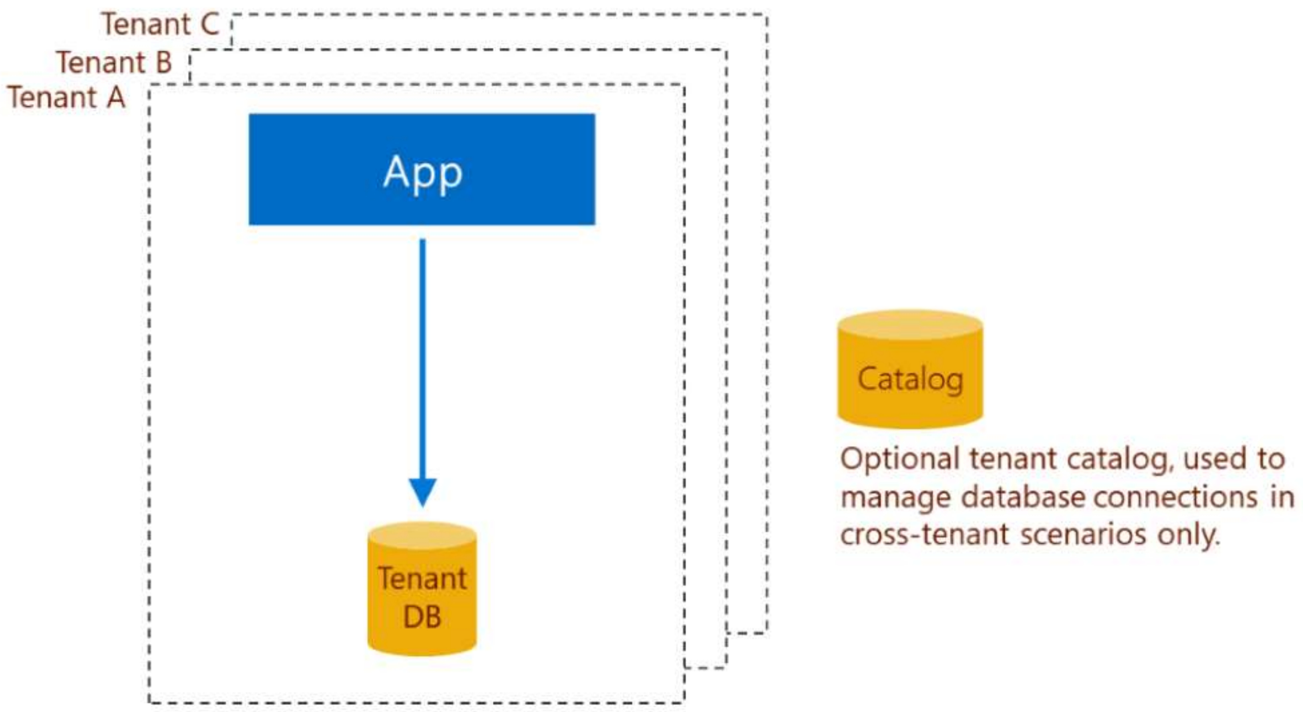
- **Scalability:**
  - Number of tenants.
  - Storage per-tenant.
  - Storage in aggregate.
  - Workload.
- **Tenant isolation:** Data isolation and performance (whether one tenant's workload impacts others).
- **Per-tenant cost:** Database costs.
- **Development complexity:**
  - Changes to schema.
  - Changes to queries (required by the pattern).
- **Operational complexity:**
  - Monitoring and managing performance.
  - Schema management.
  - Restoring a tenant.
  - Disaster recovery.
- **Customizability:** Ease of supporting schema customizations that are either tenant-specific or tenant class-specific.

The tenancy discussion is focused on the *data* layer. But consider for a moment the *application* layer. The application layer is treated as a monolithic entity. If you divide the application into many small components, your choice of tenancy model might change. You could treat some components differently than others regarding both tenancy and the storage technology or platform used.

C. Standalone single-tenant app with single-tenant database

Application level isolation

In this model, the whole application is installed repeatedly, once for each tenant. Each instance of the app is a standalone instance, so it never interacts with any other standalone instance. Each instance of the app has only one tenant, and therefore needs only one database. The tenant has the database all to itself.


Tenant C
Tenant B
Tenant A
App
Tenant DB
Catalog
Optional tenant catalog, used to manage database connections in cross-tenant scenarios only.

Each app instance is installed in a separate Azure resource group. The resource group can belong to a subscription that is owned by either the software vendor or the tenant. In either case, the vendor can manage the software for the tenant. Each application instance is configured to connect to its corresponding database.

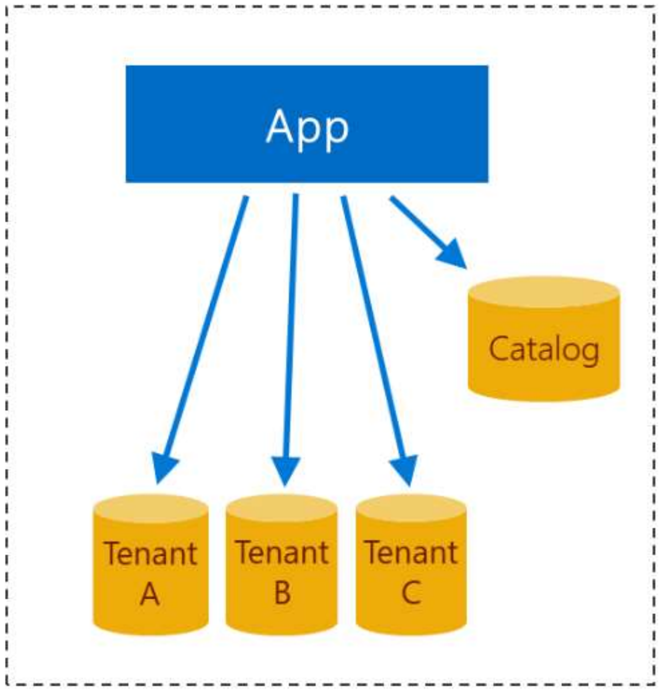
Each tenant database is deployed as a standalone database. This model provides the greatest database isolation. But the isolation requires that sufficient resources be allocated to each database to handle its peak loads. Here it matters that elastic pools cannot be used for databases deployed in different resource groups or to different subscriptions. This limitation makes this standalone single-tenant app model the most expensive solution from an overall database cost perspective.

Vendor management

The vendor can access all the databases in all the standalone app instances, even if the app instances are installed in different tenant subscriptions. The access is achieved via SQL connections. This cross-instance access can enable the vendor to centralize schema management and cross-database query for reporting or analytics purposes. If this kind of centralized management is desired, a catalog must be deployed that maps tenant identifiers to database URIs. Azure SQL Database provides a sharding library that is used together with a SQL database to provide a catalog. The sharding library is formally named the Elastic Database Client Library.

D. Multi-tenant app with database-per-tenant

This next pattern uses a multi-tenant application with many databases, all being single-tenant databases. A new database is provisioned for each new tenant. The application tier is scaled *up* vertically by adding more resources per node. Or the app is scaled *out* horizontally by adding more nodes. The scaling is based on workload, and is independent of the number or scale of the individual databases.



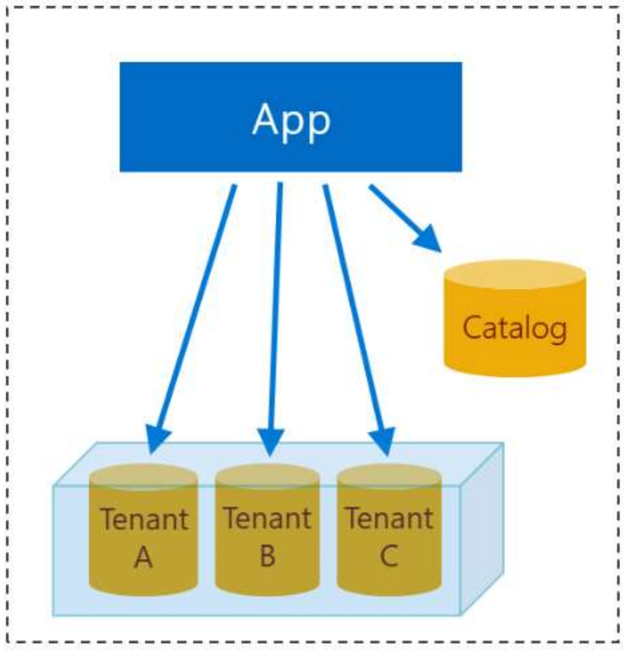
Customize for a tenant

Like the standalone app pattern, the use of single-tenant databases gives strong tenant isolation. In any app whose model specifies only single-tenant databases, the schema for any one given database can be customized and optimized for its tenant. This customization does not affect other tenants in the app. Perhaps a tenant might need data beyond the basic data fields that all tenants need. Further, the extra data field might need an index.

With database-per-tenant, customizing the schema for one or more individual tenants is straightforward to achieve. The application vendor must design procedures to carefully manage schema customizations at scale.

Elastic pools

When databases are deployed in the same resource group, they can be grouped into elastic database pools. The pools provide a cost-effective way of sharing resources across many databases. This pool option is cheaper than requiring each database to be large enough to accommodate the usage peaks that it experiences. Even though pooled databases share access to resources they can still achieve a high degree of performance isolation.



Azure SQL Database provides the tools necessary to configure, monitor, and manage the sharing. Both pool-level and database-level performance metrics are available in the Azure portal, and through Log Analytics. The metrics can give great insights into both aggregate and tenant-specific performance. Individual databases can be moved between pools to provide reserved resources to a specific tenant. These tools enable you to ensure good performance in a cost effective manner.

Operations scale for database-per-tenant

The Azure SQL Database platform has many management features designed to manage large numbers of databases at scale, such as well over 100,000 databases. These features make the database-per-tenant pattern plausible.

For example, suppose a system has a 1000-tenant database as its only one database. The database might have 20 indexes. If the system converts to having 1000 single-tenant databases, the quantity of indexes rises to 20,000. In SQL Database as part of Automatic tuning, the automatic indexing features are enabled by default. Automatic indexing manages for you all 20,000 indexes and their ongoing create and drop optimizations. These automated actions occur within an individual database, and they are not coordinated or restricted by similar actions in other databases. Automatic indexing treats indexes differently in a busy database than in a less busy database. This type of index management customization would be impractical at the database-per-tenant scale if this huge management task had to be done manually.

Other management features that scale well include the following:

- Built-in backups.
- High availability.
- On-disk encryption.
- Performance telemetry.

Automation

The management operations can be scripted and offered through a devops model. The operations can even be automated and exposed in the application.

For example, you could automate the recovery of a single tenant to an earlier point in time. The recovery only needs to restore the one single-tenant database that stores the tenant. This restore has no impact on other tenants, which confirms that management operations are at the finely granular level of each individual tenant.

E. Multi-tenant app with multi-tenant databases

Another available pattern is to store many tenants in a multi-tenant database. The application instance can have any number of multi-tenant databases. The schema of a multi-tenant database must have one or more tenant identifier columns so that the data from any given tenant can be selectively retrieved. Further, the schema might require a few tables or columns that are used by only a subset of tenants. However, static code and reference data is stored only once and is shared by all tenants.

Tenant isolation is sacrificed

Data: A multi-tenant database necessarily sacrifices tenant isolation. The data of multiple tenants is stored together in one database. During development, ensure that queries never expose data from more than one tenant. SQL Database supports row-level security, which can enforce that data returned from a query be scoped to a single tenant.

Processing: A multi-tenant database shares compute and storage resources across all its tenants. The database as a whole can be monitored to ensure it is performing acceptably. However, the Azure system has no built-in way to monitor or manage the use of these resources by an individual tenant. Therefore, the multi-tenant database carries an increased risk of encountering noisy neighbors, where the workload of one overactive tenant impacts the performance experience of other tenants in the same database. Additional application-level monitoring could monitor tenant-level performance.

Lower cost

In general, multi-tenant databases have the lowest per-tenant cost. Resource costs for a standalone database are lower than for an equivalently sized elastic pool. In addition, for scenarios where tenants need only limited storage, potentially millions of tenants could be stored in a single database. No elastic pool can contain millions of databases. However, a solution containing 1000 databases per pool, with 1000 pools, could reach the scale of millions at the risk of becoming unwieldy to manage.

Two variations of a multi-tenant database model are discussed in what follows, with the sharded multi-tenant model being the most flexible and scalable.

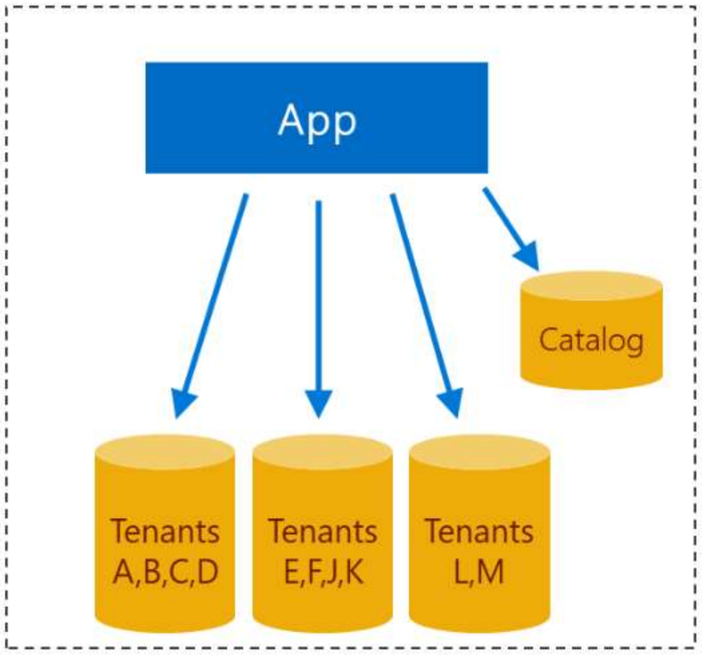
F. Multi-tenant app with a single multi-tenant database

The simplest multi-tenant database pattern uses a single standalone database to host data for all tenants. As more tenants are added, the database is scaled up with more storage and compute resources. This scale up might be all that is needed, although there is always an ultimate scale limit. However, long before that limit is reached the database becomes unwieldy to manage.

Management operations that are focused on individual tenants are more complex to implement in a multi-tenant database. And at scale these operations might become unacceptably slow. One example is a point-in-time restore of the data for just one tenant.

G. Multi-tenant app with sharded multi-tenant databases

Most SaaS applications access the data of only one tenant at a time. This access pattern allows tenant data to be distributed across multiple databases or shards, where all the data for any one tenant is contained in one shard. Combined with a multi-tenant database pattern, a sharded model allows almost limitless scale.



Manage shards

Sharding adds complexity both to the design and operational management. A catalog is required in which to maintain the mapping between tenants and databases. In addition, management procedures are required to manage the shards and the tenant population. For example, procedures must be designed to add and remove shards, and to move tenant data between shards. One way to scale is to by adding a new shard and populating it with new tenants. At other times you might split a densely populated shard into two less-densely populated shards. After several tenants have been moved or discontinued, you might merge sparsely populated shards together. The merge would result in more cost-efficient resource utilization. Tenants might also be moved between shards to balance workloads.

SQL Database provides a split/merge tool that works in conjunction with the sharding library and the catalog database. The provided app can split and merge shards, and it can move tenant data between shards. The app also maintains the catalog during these operations, marking affected tenants as offline prior to moving them. After the move, the app updates the catalog again with the new mapping, and marking the tenant as back online.

Smaller databases more easily managed

By distributing tenants across multiple databases, the sharded multi-tenant solution results in smaller databases that are more easily managed. For example, restoring a specific tenant to a prior point in time now involves restoring a single smaller database from a backup, rather than a larger database that contains all tenants. The database size, and number of tenants per database, can be chosen to balance the workload and the management efforts.

Tenant identifier in the schema

Depending on the sharding approach used, additional constraints may be imposed on the database schema. The SQL Database split/merge application requires that the schema includes the sharding key, which typically is the tenant identifier. The tenant identifier is the leading element in the primary key of all sharded tables. The tenant identifier enables the split/merge application to quickly locate and move data associated with a specific tenant.

Elastic pool for shards

Sharded multi-tenant databases can be placed in elastic pools. In general, having many single-tenant databases in a pool is as cost efficient as having many tenants in a few multi-tenant databases. Multi-tenant databases are advantageous when there are a large number of relatively inactive tenants.

H. Hybrid sharded multi-tenant database model

In the hybrid model, all databases have the tenant identifier in their schema. The databases are all capable of storing more than one tenant, and the databases can be sharded. So in the schema sense, they are all multi-tenant databases. Yet in practice some of these databases contain only one tenant. Regardless, the quantity of tenants stored in a given database has no effect on the database schema.

Move tenants around

At any time, you can move a particular tenant to its own multi-tenant database. And at any time, you can change your mind and move the tenant back to a database that contains multiple tenants. You can also assign a tenant to new single-tenant database when you provision the new database.

The hybrid model shines when there are large differences between the resource needs of identifiable groups of tenants. For example, suppose that tenants participating in a free trial are not guaranteed the same high level of performance that subscribing tenants are. The policy might be for tenants in the free trial phase to be stored in a multi-tenant database that is shared among all the free trial tenants. When a free trial tenant subscribes to the basic service level, the tenant can be moved to another multi-tenant database that might have fewer tenants. A subscriber that pays for the premium service level could be moved to its own new single-tenant database.

Pools

In this hybrid model, the single-tenant databases for subscriber tenants can be placed in resource pools to reduce database costs per tenant. This is also done in the database-per-tenant model.

I. Tenancy models compared

The following table summarizes the differences between the main tenancy models.

Measurement	Standalone app	Database-per-tenant	Sharded multi-tenant
Scale	Medium 1-100s	Very high 1-100,000s	Unlimited 1-1,000,000s
Tenant isolation	Very high	High	Low; except for any singleton tenant (that is alone in an MT db).
Database cost per tenant	High; is sized for peaks.	Low; pools used.	Lowest, for small tenants in MT DBs.
Performance monitoring and management	Per-tenant only	Aggregate + per-tenant	Aggregate; although is per-tenant only for singletons.
Development complexity	Low	Low	Medium; due to sharding.
Operational complexity	Low-High. Individually simple, complex at scale.	Low-Medium. Patterns address complexity at scale.	Low-High. Individual tenant management is complex.

Next steps

- Deploy and explore a multi-tenant Wingtip application that uses the database-per-tenant SaaS model - Azure SQL Database
- Welcome to the Wingtip Tickets sample SaaS Azure SQL Database tenancy app