MULTI-TENANT APPROACH

CONTENTS

1. DOCUMENT PURPOSE 3
2. MULTI-TENANT APP WITH DATABASE PER TENANT 3
2.1 Client 4
2.2 Server Application 4
2.3 Secure store service 4
2.4 Tenant database 4
3. CHARACTERISTICS 5
3.1 Data separation 5
3.2 Database performance and reliability 5
3.3 Implementation complexity 5
3.4 Database scalability 5
3.5 SQL vs No-SQL databases 6
3.6 Deployment and maintenance 6
3.7 Backup and restore 6
4. REFERENCES 6

FIGURES

Figure 1: Multi-tenant application with a database per tenant 3
Figure 2: Example - Tenant A database is on Database Server 1. Tenant B and Tenant C databases are sharing Database Server 2. 4
Figure 3: Example - All tenant databases are sharing Database Server 1 5

1. Document Purpose

The purpose of this document is to define and describe the multi-tenant implementation approach. Included are the main characteristics of the proposed approach, commonly known as **"multi-tenant application with database per tenant"** pattern.

2.0 Multi-tenant app with database per tenant

With the multi-tenant application with a database per tenant approach, there is one secure store that will hold the tenants secure data (like the connection string to their database, or file storage etc.). Tenant separation is achieved at the "Tenant handler" layer, where the application resolves which tenant data to use.

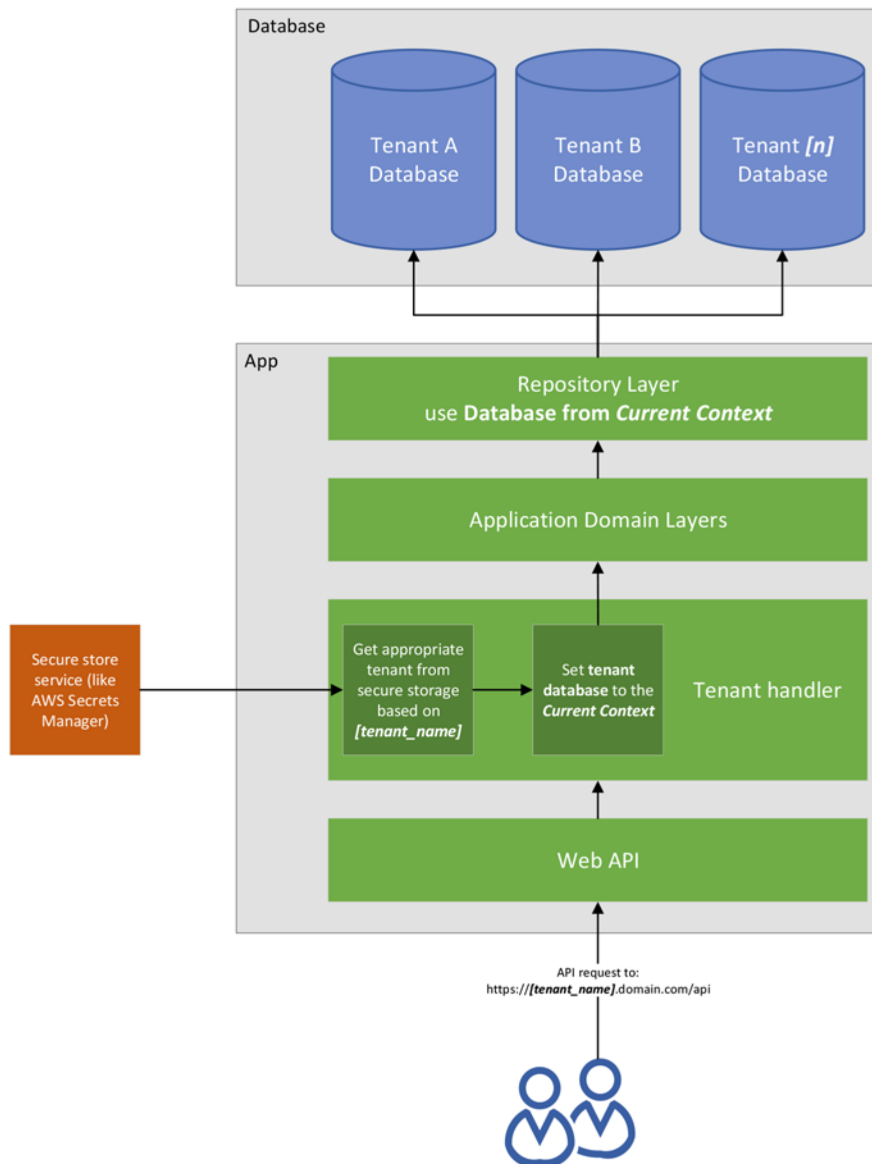

Figure 1: Multi-tenant application with a database per tenant

2.1 Client

The client application is not aware of multi-tenant implementation. This application will only use the tenant's name when accessing the server application. This is usually achieved by defining the server application's subdomain for each tenant, and the client application communicates with **[tenant_name]**.*app-domain/api*.

2.2 Server Application

With database per tenant implementation, there is one application instance for all tenants. The application is aware of the client's tenant and knows what database to use for the client's tenant. Tenant information is usually part of the client's request.

A separate layer in the application is responsible for reading the tenant-specific data (**tenant_handler** layer.) This layer is using the secure store service (like AWS secrets manager) to read the tenant-specific database connection string and database credentials, storing that information in the current context.

All other parts of the application are tenant unaware, and tenant separation is achieved at the database access (database repository) layer. The repository layer is using the information from the current context to access the tenant's specific database instance.

2.3 Secure store service

The Secure store service is used to store and serve the tenant information. This service typically stores id, unique-name, database address, and database credentials for the tenants, etc.

2.4 Tenant database

Each tenant database is responsible for storing and serving the tenant-specific applications. The application's repository layer is aware of the tenant-specific database, and it is using the information from the current context to help the application's domain layers.

Depending upon the requirements, the tenant's database can be hosted on either a shared or a separate location.

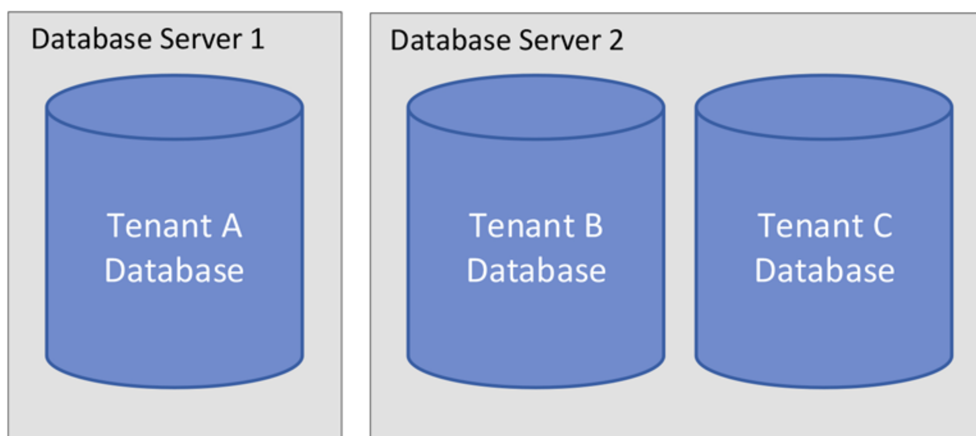

Figure 2: Example - Tenant A database is on Database Server 1. Tenant B and Tenant C databases are sharing Database Server 2.

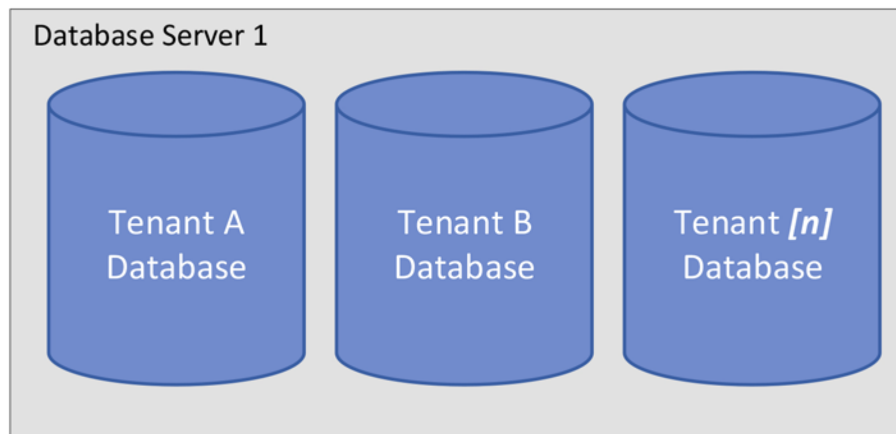

Figure 3: Example - All tenant databases are sharing Database Server 1

3.0 Characteristics

3.1 Data separation

There are a couple of items which must be considered regarding data separation:

- For good cross-tenant data separation, data is separated in the specific tenant's database, and there is no mixing of data for different tenants.
- In the case of added complexity, where report(s) need to summarize data from all tenants, usually, some additional reporting approach is implemented on top of the implementation.

3.2 Database performance and reliability

The database per tenant approach ensures better database performance. With this approach, data partitioning is implemented from the highest level (the tenants.) Also, since data is separated per tenant, one indexing level is avoided. With a multi-tenant database approach, all collections/tables will generate an index for the tenant specification field.

Because the tenant's instances are separated, if some issue arises with one tenant's database, the application will continue working for all the other tenants.

3.3 Implementation complexity

Most of the application is tenant unaware. Tenant specific functionality is isolated in the tenant-handler layer, and this information is used in the data access (data repository) layer of the application. This ensures that there will be no tenant-specific functionality across the different application domain layers.

3.4 Database scalability

For small databases, all tenants can share one database server resource. As database size and usage increase, the hardware of the database server resource can be scaled up, or a specific tenant's database can be separated onto a new instance.

3.5 SQL vs No-SQL databases

For No-SQL database engines, the process of creating a database and maintaining the database schema is generally easier and more automated. With the correct database user permissions, as data comes into the system, the application code can create both the database and the collections. Meaning that when defining a new tenant in the system, the only thing that must be done is to define the tenant's information in the main database. Then the application will know how to start working for that tenant. For generating indexes and functions on the database level, the solution will need to include procedure(s) for handling new tenants.

For SQL database engines, the process of defining a new tenant in the system will involve creating a database for the tenant. This includes having database schema scripts for generating the tenant's database schema, creating the new database for the tenant, and executing the schema scripts on the new database.

3.6 Deployment and maintenance

The deployment procedure should cover all tenant databases. To avoid any future complications, all databases must always be on the same schema version. When a new application version is released, databases changes will affect all tenant instances.

In the process of defining maintenance functions and procedures, all tenant instances will be covered. It should be noted that many tenants can result in extra work to maintain all the databases.

3.7 Backup and restore

A backup process should be defined for all tenant databases, which will result in additional work for the DB Admin and/or DevOps team. However, by having well-defined procedures for backup and restoration, these procedures can be performed on one tenant's instance at a time without affecting all the other tenants.

4.0 References

A showcase implementation of the Multi-tenant approach:

https://github.com/IT-Labs/MultiTenant

© IT Labs 4521 PGA Blvd #224, Palm Beach

+1 323 384 7368

contact@it-labs.com

www.it-labs.com