1.5 Multi Tenancy

1.5.1 What Is Multi Tenancy?

Wikipedia: "*Software **Multitenancy** refers to a software **architecture** in which a single instance of a software runs on a server and serves **multiple tenants**. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a software application is designed to provide every tenant a **dedicated share of the instance including its data**, configuration, user management, tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants*"

Multi-tenancy is used to create **SaaS** (Software as-a Service) applications (cloud computing). There are some architectures multi-tenancy:

Multiple Deployment - Multiple Database

This is **not multi tenancy** actually. But, if we run **one instance** of the application **for each** customer (tenant) with a **seperated database**, we can serve to **multiple tenants** in a single server. We just make sure that multiple instance of the application don't **conflict** with each other in same server environment.

This can be possible also for an **existing application** which is not designed as multitenant. It's easier to create such an application since the application has not aware of multitenancy. But there are setup, utilization and maintenance problems in this approach.

Single Deployment - Multiple Database

In this approach, we may run a **single instance** of the application in a server. We have a **master** (host) database to store tenant metadata (like tenant name and subdomain) and a **seperated database** for each tenant. Once we identify the **current tenant** (for example; from subdomain or from a user login form), then we can **switch** to that tenant's database to perform operations.

In this approach, application should be designed as multi-tenant in some level. But most of the application can remain independed from multi-tenancy.

We should create and maintain a **seperated database** for each tenant, this includes **database migrations**. If we have many customers with dedicated databases, it may take long time to migrate database schema in an application update. Since we have seperated database for a tenant, we can **backup** it's database seperately from other tenants. Also, we can **move** the tenant database to a stronger server if that tenant needs it.

Single Deployment - Single Database

This is the most **real multi-tenancy** architecture: We only **deploy single instance** of the application with a **single database** into a **single server**. We have a **TenantId** (or similar) field in each table (for a RDBMS) which is used to isolate a tenant's data from others.

This is easy to setup and maintain. But harder to create such an application. Because, we must prevent a Tenant to read or write other tenant data. We may add **TenantId filte**r for each database read (select) operation. Also, we may check it every write, if this entity is related to the current tenant. This is tedious and error-prone. But StudioX helps us here by using automatic data filtering.

This approach may have performance problems if we have many tenants with huge data. We may use table partitioning or other database features to overcome this problem.

Single Deployment - Hybrid Databases

We may want to store tenants in single databases normally, but want to create seperated databases for desired tenants. For example, we can store tenants with big data in their own databases, but store all other tenants in a single database.

Multiple Deployment - Single/Multiple/Hybrid Database

Finally, we may want to deploy our application to more than one server (like web farms) for a better application performance, high availability, and/or scalability. This is independent from database approach.

1.5.2 Multi-Tenancy in StudioX

StudioX can work with all scenarios described above.

Enabling Multi Tenancy

Multi-tenancy is disabled by default. We can enable it in PreInitialize of our module as shown below:

```
Configuration.MultiTenancy.IsEnabled = true;
```

Host vs Tenant

First, we should define two terms used in a multi-tenant system:

- **Tenant**: A customer which have it's own users, roles, permissions, settings... and uses the application completely isolated from other tenants. A multi-tenant application will have one or more tenants. If this is a CRM application, different tenants have also thier own accounts, contacts, products and orders. So, when we say a **'tenant user'**, we mean a user owned by a tenant.
- **Host**: Host is singleton (there is a single host). The Host is responsible to create and manage tenants. So, a **'host user'** is higher level and independent from all tenants and can control they.

Session

StudioX defines **IStudioXSession** interface to obtain current **user** and **tenant** ids. This interface is used in multi-tenancy to get current tenant's id by default. Thus, it can filter data based on current tenant's id. We can say these rules:

- If both of UserId and TenantId is null, then current user is **not logged** in to the system. So, we can not know if it's a host user or tenant user. In this case, user can not access to authorized content.
- If UserId is not null and TenantId is null, then we can know that current user is a **host user**.
- If UserId is not null and also TenantId is not null, we can know that current user is a **tenant user**.

See session documentation for more information on the session.

Determining Current Tenant

Since all tenant users use the same application, we should have a way of distinguishing the tenant of the current request. Default session implementation (ClaimsStudioXSession) uses different approaches to find the tenant related to the current request with the given order:

- If user has logged in, then gets TenantId from current claims. Claim name should contain an integer value. If it's not found in claims then the user is assumed as a host user.
- If user has not logged in, then it tries to resolve TenantId from tenant resolve contributors. There are 3 pre-defined tenant contributors and runs in given order (first success resolver wins):
    - **DomainTenantResolveContributer**: Tries to resolve tenancy name from url, generally from domain or subdomain. You can configure domain format in PreInitialize method of your module (like Configuration.Modules.StudioXWebCommon().MultiTenancy.DomainFormat = "{0}.mydomain.com";). If domain format is "{0}.mydomain.com" and current host of the request is **acme**.mydomain.com, then tenancy name is resolved as "acme". Then next step is to query ITenantStore to find the TenantId by given tenancy name. If a tenant is found, then it's resolved as the current TenantId.
    - **HttpHeaderTenantResolveContributer**: Tries to resolve TenantId from "StudioX.TenantId" header value, if present (This is a constant defined in StudioX.MultiTenancy.MultiTenancyConsts.TenantIdResolveKey).
    - **HttpCookieTenantResolveContributer**: Tries to resolve TenantId from "StudioX.TenantId" cookie value, if present (uses the same constant explained above).

If none of these attemtps can resolve a TenantId, then current requester is considered as the host. Tenant resolvers are extensible. You can add resolvers to **Configuration.MultiTenancy.Resolvers** collection, or remove an existing resolver.

One last thing on resolvers is that; resolved tenant id is cached during the same request for performance reasons. So, resolvers are executed once in a request (and only if current user has not already logged in).

Tenant Store

DomainTenantResolveContributer uses ITenantStore to find tenant id by tenancy name. Default implementation of **ITenantStore** is **NullTenantStore** which does not contain any tenant and returns null for queries. You can implement and replace it to query tenants from any data source. Module zero properly implements it to get from it's tenant manager. So, if you are using module zero, don't care about the tenant store.

Data Filters

For **multi tenant single database** approach, we must add a **TenantId** filter to get only current tenant's entities while retrieving entities from database. StudioX automatically does it when you implement one of two interfaces for your entity: **IMustHaveTenant** and **IMayHaveTenant**.

IMustHaveTenant Interface

This interface is used to distinguish entities of different tenants by defining **TenantId** property. An example entitiy that implements IMustHaveTenant:

```
public class Product : Entity, IMustHaveTenant
{
    public int TenantId { get; set; }
    public string Name { get; set; }

    //...other properties
}
```

Thus, StudioX knows that this is a tenant-specific entity and automatically isolates entities of a tenant from other tenants.

IMayHaveTenant interface

We may need to share an entity type between host and tenants. So, an entity may be owned by a tenant or the host. IMayHaveTenant interface also defines TenantId (similar to IMustHaveTenant), but nullable in this case. An example entitiy that implements IMayHaveTenant:

```
public class Product : Entity, IMayHaveTenant
{
    public int? TenantId { get; set; }
    public string Name { get; set; }

    //...other properties
}
```

We may use same Role class to store Host roles and Tenant roles. In this case, TenantId property says if this is an host entity or tenant entitiy. A null value means this is a host entity, a non-null value means this entity owned by a tenant which's Id is the TenantId.

Additional Notes

IMayHaveTenant is not common as IMustHaveTenant. For example, a Product class can not be IMayHaveTenant since a Product is related to actual application functionality, not related to managing tenants. So, use IMayHaveTenant interface carefully since it's harder to maintain a code shared by host and tenants.

When you define an entity type as IMustHaveTenant or IMayHaveTenant, always set TenantId when you create a new entity (While StudioX tries to set it from current TenantId, it may not be possible in some cases, especially for IMayHaveTenant entities). Most of times, this will be the only point you deal with TenantId properties. You don't need to explicitly write TenantId filter in Where conditions while writing LINQ, since it will be automatically filtered.

Switching Between Host and Tenants

While working on a multitenant application database, we should know the **current tenant**. By default, it's obtained from IStudioXSession (as described before). We can change this behaviour and switch to other tenant's database. Example:

```
public class ProductService : ITransientDependency
{
    private readonly IRepository<Product> productRepository;
    private readonly IUnitOfWorkManager unitOfWorkManager;

    public ProductService(
        IRepository<Product> productRepository,
        UnitOfWorkManager unitOfWorkManager)
    {
        this.productRepository = productRepository;
        this.unitOfWorkManager = unitOfWorkManager;
    }

    [UnitOfWork]
    public virtual List<Product> GetProducts(int tenantId)
    {
        using (this.unitOfWorkManager.Current.SetTenantId(tenantId))
        {
            return this.productRepository.GetAllList();
        }
    }
}
```

SetTenantId ensures that we are working on given tenant data, independent from database architecture:

- If given tenant has a dedicated database, it switches to that database and gets products from it.
- If given tenant has not a dedicated database (single database approach, for example), it adds automatic TenantId filter to query get only that tenant's products.

If we don't use SetTenantId, it gets tenantId from session, as said before. There are some notes and best practices here:

- **Use SetTenantId(null)** to switch to the host.
- Use SetTenantId within a **using** block as in the example if there is not a special case. Thus, it automatically restore tenantId at the end of the using block and the code calls GetProducts method works as before.
- You can use SetTenantId in **nested blocks** if it's needed.
- Since **unitOfWorkManager.Current** only available in a unit of work, be sure that your code runs in a uow.