Filter...

latest (v7.1)

ABP Framework

Overall

Introduction
Tutorials & Articles
NLayer Architecture
Module System
Startup Configuration
Multi-Tenancy
OWIN Integration
Debugging
API Reference

Common Structures

Dependency Injection
Session
Caching
Logging
Setting Management
Timing
Object To Object Mapping (and AutoMapper Integration)
Email Sending (and MailKit Integration)

Domain Layer

Entities
Multi-Lingual Entities
Value Objects
Repositories
Domain Services
Specifications
Unit Of Work
Domain Events (EventBus)
Data Filters
Dynamic Parameter System
Object Comparators

Application Layer

Application Services
Data Transfer Objects
Validating Data Transfer Objects

In this document

What Is Multi-Tenancy?
Database & Deployment Architectures
Multi-Tenancy in ASP.NET Boilerplate
Enabling Multi-Tenancy
Ignore Feature Check For Host Users
Host vs Tenant
Session
Determining Current Tenant
Data Filters
Switching Between Host and Tenants

Edit on GitHub

What Is Multi-Tenancy?

*"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"* (Wikipedia)

In short, multi-tenancy is a technique that is used to create **SaaS** (Software as-a Service) applications.

Database & Deployment Architectures

There are some different multi-tenant database & deployment approaches:

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 **separated database**, we can serve **multiple tenants** on a single server. We just have to make sure that multiple instances of the application don't **conflict** with each other on the same server environment.

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

Single Deployment - Multiple Database

In this approach, we run a **single instance** of the application on a server. We have a **master** (host) database to store tenant metadata (like tenant name and subdomain) and a **separate 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, the application should be designed as multi-tenant at some level, but most of the application can remain independent from it.

We create and maintain a **separate database** for each tenant, this includes **database migrations**. If we have many customers with dedicated databases, it may take a long time to migrate the database schema during an application update. Since we have a separated database for each tenant, we can **backup** its database separately from other tenants. We can also **move** the tenant database to a stronger server if that tenant needs it.

Single Deployment - Single Database

This is the **most ideal multi-tenancy** architecture: We only deploy a **single instance** of the application with a **single database** on to 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 type of application is easy to setup and maintain, but harder to create. This is because we must prevent a Tenant from reading or writing to other tenant data. We may add a **TenantId filter** for each database read (select) operation. We may also check it every time we write, to see if this entity is related to the **current tenant**. This is tedious and error-prone. However, ASP.NET Boilerplate helps us here by using **automatic data filtering**.

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

Single Deployment - Hybrid Databases

We may want to store tenants in a single databases normally, but may want to create a separate database 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 better application performance, high availability, and/or scalability. This is independent from the database approach.

Multi-Tenancy in ASP.NET Boilerplate

ASP.NET Boilerplate can work with all the scenarios described above.

Enabling Multi-Tenancy

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

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

Note: Multi-tenancy is enabled in both ASP.NET Core and ASP.NET MVC 5.x startup templates.

Ignore Feature Check For Host Users

There is another configuration to ignore feature check for host users. We can enable it in PreInitialize method of our module as shown below:

```
Configuration.MultiTenancy.IgnoreFeatureCheckForHostUsers = true;
```

Note: `IgnoreFeatureCheckForHostUsers` default value is `false`;

Host vs Tenant

We define two terms used in a multi-tenant system:

- **Tenant**: A customer which has its 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 also have their own accounts, contacts, products and orders. So when we say a '**tenant user**', we mean a user owned by a tenant.

- **Host**: The Host is singleton (there is a single host). The Host is responsible for creating and managing tenants. A '**host user**' is at a higher level and independent from all tenants and can control them.

Session

ASP.NET Boilerplate defines the **IAbpSession** interface to obtain the current **user** and **tenant** ids. This interface is used in multi-tenancy to get the current tenant's id by default. Thus, it can filter data based on the current tenant's id. Here are the rules:

- If both the UserId and the TenantId is null, then the current user is **not logged in** to the system. We can not know if it's a host user or tenant user. In this case, the user can not access authorized content.
- If the UserId is not null and the TenantId is null, then we know that the current user is a **host user**.
- If the UserId is not null and the TenantId is not null, we know that the current user is a **tenant user**.
- If the UserId is null but the TenantId is not null, that means we know the current tenant, but the current request is not authorized (user did not login). See the next section to understand how the current tenant is determined.

See the session documentation for more information.

Determining Current Tenant

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

1. If the user is logged in, it gets the TenantId from current claims. Claim name is *http://www.aspnetboilerplate.com/identity/claims/tenantId* and should contain an integer value. If it's not found in claims then the user is assumed to be a *host* user.
2. If the user has not logged in, then it tries to resolve the TenantId from the *tenant resolve contributor*s. There are 3 pre-defined tenant contributors and are run in a given order (first successful resolver 'wins'):
    1. **DomainTenantResolveContributor**: Tries to resolve tenancy name from an url, generally from a domain or subdomain. You can configure the domain format in the PreInitialize method of your module (like Configuration.Modules.AbpWebCommon().MultiTenancy.DomainFormat = "{0}.mydomain.com";). If the domain format is "{0}.mydomain.com" and the current host of the request is ***acme*.mydomain.com**, then the tenancy name is resolved as "acme". The next step is to query ITenantStore to find the TenantId by the given tenancy name. If a tenant is found, then it's resolved as the current TenantId.
    2. **HttpHeaderTenantResolveContributor**: Tries to resolve TenantId from an "Abp.TenantId" header value, if present. This is a constant defined in Abp.MultiTenancy.MultiTenancyConsts.TenantIdResolveKey.
    3. **HttpCookieTenantResolveContributor**: Tries to resolve the TenantId from an "Abp.TenantId" cookie value, if present. This uses the same constant explained above.

By default, ASP.NET Boilerplate uses "Abp.TenantId" to find TenantId from Cookie or Request Headers. You can change it using multi-tenancy configuration:

Copy

```
Configuration.MultiTenancy.TenantIdResolveKey = "Abp-TenantId";
```

You also need to configure it on the client side:

```
abp.multiTenancy.tenantIdCookieName = 'Abp-TenantId';
```

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

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

Tenant Store

The **DomainTenantResolveContributor** uses ITenantStore to find the tenant id by tenancy name. The 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 by getting it from its tenant manager. So if you are using Module Zero, you don't need to worry about the tenant store.

Data Filters

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

IMustHaveTenant Interface

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

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

    public string Name { get; set; }

    //...other properties
}
```

This way, ASP.NET Boilerplate knows that this is a tenant-specific entity and automatically isolates the entities of a tenant from other tenants.

IMayHaveTenant interface

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

```
public class Role : Entity, IMayHaveTenant
{
    public int? TenantId { get; set; }

    public string RoleName { get; set; }

    //...other properties
}
```

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

Additional Notes

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

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

Switching Between Host and Tenants

While working on a multi-tenant application database, we can get the **current tenant**. By default, it's obtained from the IAbpSession (as described before). We can change this behavior and switch to another tenant's database. Example:

Copy

```
public class ProductService : ITransientDependency
{
    private readonly IRepository<Product> _productRepository;
    private readonly IUnitOfWorkManager _unitOfWorkManager;

    public ProductService(IRepository<Product> productRepository, IUnitOfWorkManager
unitOfWorkManager)
    {
        _productRepository = productRepository;
        _unitOfWorkManager = unitOfWorkManager;
    }

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

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

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

If we don't use SetTenantId, it gets the tenantId from the session. There are some guidelines 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. This way, it automatically restores the tenantId at the end of the using block and the code calling the GetProducts method works as before.
- You can use SetTenantId in **nested blocks** if it's needed.
- Since **_unitOfWorkManager.Current** is only available in a unit of work, be sure that your code runs in a UOW.