Data Transfer Objects

Introduction

Data Transfer Objects (DTO) are used to transfer data between the **Application Layer** and the **Presentation Layer** or other type of clients.

Typically, an application service is called from the presentation layer (optionally) with a **DTO** as the parameter. It uses domain objects to **perform some specific business logic** and (optionally) returns a DTO back to the presentation layer. Thus, the presentation layer is completely **isolated** from domain layer.

The Need for DTOs

> **You can skip this section** if you feel that you know and confirm the benefits of using DTOs.

At first, creating a DTO class for each application service method can be seen as tedious and time-consuming work. However, they can save your application if you correctly use them. Why & how?

Abstraction of the Domain Layer

DTOs provide an efficient way of **abstracting domain objects** from the presentation layer. In effect, your **layers** are correctly separated. If you want to change the presentation layer completely, you can continue with the existing application and domain layers. Alternatively, you can re-write your domain layer, completely change the database schema, entities and O/RM framework, all without changing the presentation layer. This, of course, is as long as the contracts (method signatures and DTOs) of your application services remain unchanged.

Data Hiding

Say you have a `User` entity with the properties Id, Name, EmailAddress and Password. If a `GetAllUsers()` method of a `UserAppService` returns a `List<User>`, anyone can access the passwords of all your users, even if you do not show it on the screen. It's not just about security, it's about data hiding. Application services should return only what it needs by the presentation layer (or client). Not more, not less.

Serialization & Lazy Load Problems

When you return data (an object) to the presentation layer, it's most likely serialized. For example, in a REST API that returns JSON, your object will be serialized to JSON and sent to the client. Returning an Entity to the presentation layer can be problematic in that regard, especially if you are using a relational database and an ORM provider like Entity Framework Core. How?

In a real-world application, your entities may have references to each other. The `User` entity can have a reference to it's `Role` s. If you want to serialize `User`, its `Role` s are also serialized. The `Role` class may have a `List<Permission>` and the `Permission` class can has a reference to a

`PermissionGroup` class and so on... Imagine all of these objects being serialized at once. You could easily and accidentally serialize your whole database! Also, if your objects have circular references, they may **not** be serialized at all.

What's the solution? Marking properties as `NonSerialized`? No, you can not know when it should be serialized and when it shouldn't be. It may be needed in one application service method, and not needed in another. Returning safe, serializable, and specially designed DTOs is a good choice in this situation.

Almost all O/RM frameworks support lazy-loading. It's a feature that loads entities from the database when they're needed. Say a `User` class has a reference to a `Role` class. When you get a `User` from the database, the `Role` property (or collection) is not filled. When you first read the `Role` property, it's loaded from the database. So, if you return such an Entity to the presentation layer, it will cause it to retrieve additional entities from the database by executing additional queries. If a serialization tool reads the entity, it reads all properties recursively and again your whole database can be retrieved (if there are relations between entities).

More problems can arise if you use Entities in the presentation layer. **It's best not to reference the domain/business layer assembly in the presentation layer.**

If you are convinced about using DTOs, we can continue to what ABP Framework provides and suggests about DTOs.

> ABP doesn't force you to use DTOs, however using DTOs is **strongly suggested as a best practice**.

Standard Interfaces & Base Classes

A DTO is a simple class that has no dependency and you can design it in any way. However, ABP introduces some **interfaces** to determine the **conventions** for naming **standard properties** and **base classes** to **don't repeat yourself** while declaring **common properties**.

None of them are required, but using them **simplifies and standardizes** your application code.

Entity Related DTOs

You typically create DTOs corresponding to your entities, which results similar classes to your entities. ABP Framework provides some base classes to simplify while creating such DTOs.

EntityDto

`IEntityDto<TKey>` is a simple interface that only defines an `Id` property. You can implement it or inherit from the `EntityDto<TKey>` for your DTOs that matches to an entity.

Example:

Share on :

In this document

In this document

```
using System;
using Volo.Abp.Application.Dtos;

namespace AbpDemo
{
    public class ProductDto : EntityDto<Guid>
    {
        public string Name { get; set; }
        //...
    }
}
```

Audited DTOs

If your entity inherits from audited entity classes (or implements auditing interfaces), you can use the following base classes to create your DTOs:

- `CreationAuditedEntityDto`
- `CreationAuditedEntityWithUserDto`
- `AuditedEntityDto`
- `AuditedEntityWithUserDto`
- `FullAuditedEntityDto`
- `FullAuditedEntityWithUserDto`

Extensible DTOs

If you want to use the object extension system for your DTOs, you can use or inherit from the following DTO classes:

- `ExtensibleObject` implements the `IHasExtraProperties` (other classes inherits this class).
- `ExtensibleEntityDto`
- `ExtensibleCreationAuditedEntityDto`
- `ExtensibleCreationAuditedEntityWithUserDto`
- `ExtensibleAuditedEntityDto`
- `ExtensibleAuditedEntityWithUserDto`
- `ExtensibleFullAuditedEntityDto`
- `ExtensibleFullAuditedEntityWithUserDto`

List Results

It is common to return a list of DTOs to the client. `IListResult<T>` interface and `ListResultDto<T>` class is used to make it standard.

The definition of the `IListResult<T>` interface:

```
public interface IListResult<T>
{
    IReadOnlyList<T> Items { get; set; }
}
```

Example: Return a list of products

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Volo.Abp.Application.Dtos;
using Volo.Abp.Application.Services;
using Volo.Abp.Domain.Repositories;

namespace AbpDemo
{
    public class ProductAppService : ApplicationService
    {
        private readonly IRepository<Product, Guid> _pr

        public ProductAppService(IRepository<Product, G
        {
            _productRepository = productRepository;
        }

        public async Task<ListResultDto<ProductDto>> Ge
        {
            //Get entities from the repository
            List<Product> products = await _productRepo

            //Map entities to DTOs
            List<ProductDto> productDtos =
                ObjectMapper.Map<List<Product>, List<Pr

            //Return the result
            return new ListResultDto<ProductDto>(produc
        }
    }
}
```

Share on :

In this document

You could simply return the `productDtos` object (and change the method return type) and it has nothing wrong. Returning a `ListResultDto` makes your `List<ProductDto>` wrapped into another object as an `Items` property. This has one advantage: You can later add more properties to your return value without breaking your remote clients (when they get the value as a JSON result). So, it is especially suggested when you are developing reusable application modules.

Paged & Sorted List Results

It is more common to request a paged list from server and return a paged list to the client. ABP defines a few interface and classes to standardize it:

Input (Request) Types

The following interfaces and classes is to standardize the input sent by the clients.

- `ILimitedResultRequest` : Defines a `MaxResultCount` (`int`) property to request a limited result from the server.
- `IPagedResultRequest` : Inherits from the `ILimitedResultRequest` (so it inherently has the `MaxResultCount` property) and defines a `SkipCount` (`int`) to declare the skip count while requesting a paged result from the server.
- `ISortedResultRequest` : Defines a `Sorting` (`string`) property to request a sorted result from the server. Sorting value can be

4.1 (latest) English

Filter topics

- Getting Started
- Startup Templates
- Tutorials
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

"*Name*", "*Name DESC*", "*Name ASC, Age DESC*"... etc.

- `IPagedAndSortedResultRequest` inherits from both of the `IPagedResultRequest` and `ISortedResultRequest`, so has `MaxResultCount`, `SkipCount` and `Sorting` properties.

Instead of implementing the interfaces manually, it is suggested to inherit one of the following base DTO classes:

- `LimitedResultRequestDto` implements `ILimitedResultRequest`.
- `PagedResultRequestDto` implements `IPagedResultRequest` (and inherits from the `LimitedResultRequestDto`).
- `PagedAndSortedResultRequestDto` implements `IPagedAndSortedResultRequest` (and inherit from the `PagedResultRequestDto`).

Max Result Count

`LimitedResultRequestDto` (and inherently the others) limits and validates the `MaxResultCount` by the following rules;

- If the client doesn't set `MaxResultCount`, it is assumed as **10** (the default page size). This value can be changed by setting the `LimitedResultRequestDto.DefaultMaxResultCount` static property.
- If the client sends `MaxResultCount` greater than **1,000**, it produces a **validation error**. It is important to protect the server from abuse of the service. If you want, you can change this value by setting the `LimitedResultRequestDto.MaxMaxResultCount` static property.

Static properties suggested to be set on application startup since they are static (global).

Output (Response) Types

The following interfaces and classes is to standardize the output sent to the clients.

- `IHasTotalCount` defines a `TotalCount` (`long`) property to return the total count of the records in case of paging.
- `IPagedResult<T>` inherits from the `IListResult<T>` and `IHasTotalCount`, so it has the `Items` and `TotalCount` properties.

Instead of implementing the interfaces manually, it is suggested to inherit one of the following base DTO classes:

- `PagedResultDto<T>` inherits from the `ListResultDto<T>` and also implements the `IPagedResult<T>`.

Example: Request a paged & sorted result from server and return a paged list

Share on :

In this document

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Dynamic.Core;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Volo.Abp.Application.Dtos;
using Volo.Abp.Application.Services;
using Volo.Abp.Domain.Repositories;

namespace AbpDemo
{
    public class ProductAppService : ApplicationService
    {
        private readonly IRepository<Product, Guid> _pr

        public ProductAppService(IRepository<Product, G
        {
            _productRepository = productRepository;
        }

        public async Task<PagedResultDto<ProductDto>> G
            PagedAndSortedResultRequestDto input)
        {
            //Create the query
            var query = _productRepository
                .OrderBy(input.Sorting)
                .Skip(input.SkipCount)
                .Take(input.MaxResultCount);

            //Get total count from the repository
            var totalCount = await query.CountAsync();

            //Get entities from the repository
            List<Product> products = await query.ToList

            //Map entities to DTOs
            List<ProductDto> productDtos =
                ObjectMapper.Map<List<Product>, List<Pr

            //Return the result
            return new PagedResultDto<ProductDto>(total
        }
    }
}
```

ABP Framework also defines a `PageBy` extension method (that is compatible with the `IPagedResultRequest`) that can be used instead of `Skip` + `Take` calls:

```
var query = _productRepository
    .OrderBy(input.Sorting)
    .PageBy(input);
```

In this document

> Notice that we added `Volo.Abp.EntityFrameworkCore` package to the project to be able to use the `ToListAsync` and `CountAsync` methods since they are not included in the standard LINQ, but defined by the Entity Framework Core.

See also the repository documentation to if you haven't understood the example code.

Related Topics

Validation

Inputs of application service methods, controller actions, page model inputs... are automatically validated. You can use the standard data annotation attributes or a custom validation method to perform the validation.

See the validation document for more.

Object to Object Mapping

When you create a DTO that is related to an entity, you generally need to map these objects. ABP provides an object to object mapping system to simplify the mapping process. See the following documents:

- Object to Object Mapping document covers all the features.
- Application Services document provides a full example.

Best Practices

You are free to design your DTO classes. However, there are some best practices & suggestions that you may want to follow.

Common Principles

- DTOs should be **well serializable** since they are generally serialized and deserialized (to JSON or other format). It is suggested to have an empty (parameterless) public constructor if you have another constructor with parameter(s).
- DTOs **should not contain any business logic**, except some formal validation code.
- Do not inherit DTOs from entities and **do not reference to entities**. The application startup template already prevents it by separating the projects.
- If you use an auto object to object mapping library, like AutoMapper, enable the **mapping configuration validation** to prevent potential bugs.

Input DTO Principles

- Define only the **properties needed** for the use case. Do not include properties not used for the use case, which confuses developers if you do so.

- **Don't reuse** input DTOs among different application service methods. Because, different use cases will need to and use different properties of the DTO which results some properties are not used

in some cases and that makes harder to understand and use the services and causes potential bugs in the future.

In this document

Output DTO Principles

- You can **reuse output DTOs** if you **fill all the properties** on all the cases.