Articles Tutorials

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

Edit on GitHub

In this document

Data Transfer Objects are used to transfer data between the **Application Layer** and the **Presentation Layer**.

The Presentation Layer calls an Application Service method with a Data Transfer Object (**DTO**). The application service then uses these domain objects to perform some specific business logic, and then finally returns a DTO back to the presentation layer. Thus, the Presentation layer is completely isolated from the Domain layer. In an ideally layered application, the presentation layer never works with domain objects, (Repositories, or Entities...).

The need for 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?

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 the GetAllUsers() method of UserAppService returns a List<User>, anyone can see 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 to the presentation layer what it needs. 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 an MVC method that returns JSON, your object can be serialized to JSON and sent to the client. Returning an Entity to the presentation layer can be problematic in that regard. How?

In a real-world application, your entities will have references to each other. The User entity can have a reference to it's Roles. If you want to serialize User, its Roles are also serialized. The Role class may have a List<Permission> and the Permission class can have 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 can **not** be serialized.

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 other. 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 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. If a serialization tool reads the entity, it reads all properties recursively and again your whole database can be retrieved (if there are suitable, related properties 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.

DTO conventions & validation

ASP.NET Boilerplate strongly supports DTOs. It provides conventional classes, interfaces, and standardizes DTO naming and usage conventions. When you write your code as described here in the documentation, ASP.NET Boilerplate will easily automate some common tasks.

Example

Here's a prime example. Say that we want to develop an application service method that is used to **search people** by name and then return a **list of people**. In that case, we may have a **Person** entity as shown below:

Copy
```
public class Person : Entity
{
    public virtual string Name { get; set; }
    public virtual string EmailAddress { get; set; }
    public virtual string Password { get; set; }
}
```

We then define an interface for our application service:

Copy
```
public interface IPersonAppService : IApplicationService
{
    SearchPeopleOutput SearchPeople(SearchPeopleInput input);
}
```

ASP.NET Boilerplate suggests naming input/output parameters as MethodName**Input** and MethodName**Output** and defining a separated input and output DTO for every application service method. Even if your method only takes and returns **one** parameter, it's better to create a DTO class. In turn, your code will be more extensible. You can add more properties later without changing the signature of your method and without breaking existing client applications.

Your method can return **void** if there is no return value. It will not break existing applications if you add a return value later. If your method does not use any arguments, you do not have to define an input DTO. Keep in mind that it maybe better to write an input DTO class if you think that you'll add parameters in the future. This is up to you.

Here's how the input and output DTO classes are defined in this example:

Copy

```
public class SearchPeopleInput
{
    [StringLength(40, MinimumLength = 1)]
    public string SearchedName { get; set; }
}

public class SearchPeopleOutput
{
    public List<PersonDto> People { get; set; }
}

public class PersonDto : EntityDto
{
    public string Name { get; set; }
    public string EmailAddress { get; set; }
}
```

ASP.NET Boilerplate **automatically validates** the input before the execution of a method. It's similar to ASP.NET MVC's model validation, but note that the application service is not a Controller, it's a plain old C# class. ASP.NET Boilerplate makes an interception and checks the input automatically. See the DTO validation document for more info.

EntityDto is a simple class that declares an **Id property**, since they are common for most entities. It has a generic version if your entity's primary key is not int. You don't have to use it, but it can be better to define an Id property.

As you can see, **PersonDto** does not include a Password property since it's not needed for the presentation layer. It can be dangerous to send peoples' passwords to the presentation layer! If a JavaScript client requested it, anyone can easily grab the passwords from the result.

Let's implement **IPersonAppService** before go further:

Copy

```
public class PersonAppService : IPersonAppService
{
    private readonly IPersonRepository _personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
    }

    public SearchPeopleOutput SearchPeople(SearchPeopleInput input)
    {
        //Get entities
        var peopleEntityList = _personRepository.GetAllList(person =>
person.Name.Contains(input.SearchedName));

        //Convert to DTOs
        var peopleDtoList = peopleEntityList
            .Select(person => new PersonDto
                            {
                                Id = person.Id,
                                Name = person.Name,
                                EmailAddress = person.EmailAddress
                            }).ToList();

        return new SearchPeopleOutput { People = peopleDtoList };
    }
}
```

Here we get entities from the database, **convert** them to DTOs and then return an output. Notice that we didn't validate the input? ASP.NET Boilerplate **validate**s it. It even checks **if input parameter is null** and if so, an exception is thrown. This saves us from writing guard clauses in every method!

You're probably thinking, "Won't I have to write a whole bunch of code to move data between the Person entity and the PersonDto object?" It's really a tedious work! The Person entity could have many more properties.

Auto mapping between DTOs and entities

Fortunately, there are some tools that make this very easy! **AutoMapper** is one of them. See the AutoMapper Integration document to learn how to use it in ASP.NET Boilerplate.

Helper interfaces and classes

ASP.NET Boilerplate provides some helper interfaces that can be implemented to standardize common DTO property names.

ILimitedResultRequest defines a **MaxResultCount** property. This way, you can implement it in your input DTOs to standardize the limiting result set.

IPagedResultRequest extends **ILimitedResultRequest** by adding **SkipCount**. Let's implement this interface in SearchPeopleInput for paging:

Copy

```
public class SearchPeopleInput : IPagedResultRequest
{
    [StringLength(40, MinimumLength = 1)]
    public string SearchedName { get; set; }

    public int MaxResultCount { get; set; }
    public int SkipCount { get; set; }
}
```

As a result of a paged request, you can return an output DTO that implements **IHasTotalCount**. Naming standardization helps us to create re-usable code and conventions. Check out the interfaces and classes under the **Abp.Application.Services.Dto** namespace for more info.