In this document

Application Services

Application services are used to implement the **use cases** of an application. They are used to **expose domain logic to the presentation layer**.

An Application Service is called from the presentation layer (optionally) with a **DTO (Data Transfer Object)** 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.

Example

Book Entity

Assume that you have a `Book` entity (actually, an aggregate root) defined as shown below:

Share on :

In this document

```
public class Book : AggregateRoot<Guid>
{
    public const int MaxNameLength = 128;

    public virtual string Name { get; protected set; }

    public virtual BookType Type { get; set; }

    public virtual float? Price { get; set; }

    protected Book()
    {

    }

    public Book(Guid id, [NotNull] string name, BookTyp
    {
        Id = id;
        Name = CheckName(name);
        Type = type;
        Price = price;
    }

    public virtual void ChangeName([NotNull] string nam
    {
        Name = CheckName(name);
    }

    private static string CheckName(string name)
    {
        if (string.IsNullOrWhiteSpace(name))
        {
            throw new ArgumentException($"name can not
        }

        if (name.Length > MaxNameLength)
        {
            throw new ArgumentException($"name can not
        }

        return name;
    }
}
```

- `Book` entity has a `MaxNameLength` that defines the maximum length of the `Name` property.
- `Book` constructor and `ChangeName` method to ensure that the `Name` is always a valid value. Notice that `Name` 's setter is not `public` .

> ABP does not force you to design your entities like that. It just can have public get/set for all properties. It's your decision to full implement DDD practices.

IBookAppService Interface

In ABP, an application service should implement the `IApplicationService` interface. It's good to create an interface for each application service:

```
public interface IBookAppService : IApplicationService
{
    Task CreateAsync(CreateBookDto input);
}
```

A Create method will be implemented as the example. `CreateBookDto` is defined like that:

```
public class CreateBookDto
{
    [Required]
    [StringLength(Book.MaxNameLength)]
    public string Name { get; set; }

    public BookType Type { get; set; }

    public float? Price { get; set; }
}
```

See data transfer objects document for more about DTOs.

BookAppService (Implementation)

```
public class BookAppService : ApplicationService, IBook
{
    private readonly IRepository<Book, Guid> _bookRepos

    public BookAppService(IRepository<Book, Guid> bookR
    {
        _bookRepository = bookRepository;
    }

    public async Task CreateAsync(CreateBookDto input)
    {
        var book = new Book(
            GuidGenerator.Create(),
            input.Name,
            input.Type,
            input.Price
        );

        await _bookRepository.InsertAsync(book);
    }
}
```

- `BookAppService` inherits from the `ApplicationService` base class. It's not required, but the `ApplicationService` class provides helpful properties for common application service requirements like `GuidGenerator` used in this service. If we didn't inherit from it, we would need to inject the `IGuidGenerator` service manually (see guid generation document).
- `BookAppService` implements the `IBookAppService` as expected.

Share on :

In this document

- `BookAppService` injects `IRepository<Book, Guid>` (see repositories) and uses it inside the `CreateAsync` method to insert a new entity to the database.
- `CreateAsync` uses the constructor of the `Book` entity to create a new book from the properties of given `input` .

Data Transfer Objects

Application services gets and returns DTOs instead of entities. ABP does not force this rule. However, exposing entities to presentation layer (or to remote clients) have significant problems and not suggested.

See the DTO documentation for more.

Object to Object Mapping

The `CreateAsync` method above manually creates a `Book` entity from given `CreateBookDto` object. Because the `Book` entity enforces it (we designed it like that).

However, in many cases, it's very practical to use **auto object mapping** to set properties of an object from a similar object. ABP provides an object to object mapping infrastructure to make this even easier.

Object to object mapping provides abstractions and it is implemented by the AutoMapper library by default.

Let's create another method to get a book. First, define the method in the `IBookAppService` interface:

```
public interface IBookAppService : IApplicationService
{
    Task CreateAsync(CreateBookDto input);

    Task<BookDto> GetAsync(Guid id); //New method
}
```

`BookDto` is a simple DTO class defined as below:

```
public class BookDto
{
    public Guid Id { get; set; }

    public string Name { get; set; }

    public BookType Type { get; set; }

    public float? Price { get; set; }
}
```

AutoMapper requires to create a mapping profile class. Example:

```
public class MyProfile : Profile
{
    public MyProfile()
    {
        CreateMap<Book, BookDto>();
    }
}
```

You should then register profiles using the `AbpAutoMapperOptions`:

```
[DependsOn(typeof(AbpAutoMapperModule))]
public class MyModule : AbpModule
{
    public override void ConfigureServices(ServiceConfi
    {
        Configure<AbpAutoMapperOptions>(options =>
        {
            //Add all mappings defined in the assembly
            options.AddMaps<MyModule>();
        });
    }
}
```

`AddMaps` registers all profile classes defined in the assembly of the given class, typically your module class. It also registers for the attribute mapping.

Then you can implement the `GetAsync` method as shown below:

```
public async Task<BookDto> GetAsync(Guid id)
{
    var book = await _bookRepository.GetAsync(id);
    return ObjectMapper.Map<Book, BookDto>(book);
}
```

See the object to object mapping document for more.

Validation

Inputs of application service methods are automatically validated (like ASP.NET Core controller actions). You can use the standard data annotation attributes or a custom validation method to perform the validation. ABP also ensures that the input is not null.

See the validation document for more.

Authorization

It's possible to use declarative and imperative authorization for application service methods.

See the authorization document for more.

Share on :

In this document

Share on :

In this document

CRUD Application Services

If you need to create a simple **CRUD application service** which has Create, Update, Delete and Get methods, you can use ABP's **base classes** to easily build your services. You can inherit from the `CrudAppService`.

Example

Create an `IBookAppService` interface inheriting from the `ICrudAppService` interface.

```
public interface IBookAppService :
    ICrudAppService< //Defines CRUD methods
        BookDto, //Used to show books
        Guid, //Primary key of the book entity
        PagedAndSortedResultRequestDto, //Used for pagi
        CreateUpdateBookDto, //Used to create a new boo
        CreateUpdateBookDto> //Used to update a book
{
}
```

`ICrudAppService` has generic arguments to get the primary key type of the entity and the DTO types for the CRUD operations (it does not get the entity type since the entity type is not exposed to the clients use this interface).

> Creating interface for an application service is a good practice, but not required by the ABP Framework. You can skip the interface part.

`ICrudAppService` declares the following methods:

```
public interface ICrudAppService<
    TEntityDto,
    in TKey,
    in TGetListInput,
    in TCreateInput,
    in TUpdateInput>
    : IApplicationService
    where TEntityDto : IEntityDto<TKey>
{
    Task<TEntityDto> GetAsync(TKey id);

    Task<PagedResultDto<TEntityDto>> GetListAsync(TGetL

    Task<TEntityDto> CreateAsync(TCreateInput input);

    Task<TEntityDto> UpdateAsync(TKey id, TUpdateInput

    Task DeleteAsync(TKey id);
}
```

DTO classes used in this example are `BookDto` and `CreateUpdateBookDto`:

```
public class BookDto : AuditedEntityDto<Guid>
{
    public string Name { get; set; }

    public BookType Type { get; set; }

    public float Price { get; set; }
}

public class CreateUpdateBookDto
{
    [Required]
    [StringLength(128)]
    public string Name { get; set; }

    [Required]
    public BookType Type { get; set; } = BookType.Undef

    [Required]
    public float Price { get; set; }
}
```

Profile class of DTO class.

```
public class MyProfile : Profile
{
    public MyProfile()
    {
        CreateMap<Book, BookDto>();
        CreateMap<CreateUpdateBookDto, Book>();
    }
}
```

- `CreateUpdateBookDto` is shared by create and update operations, but you could use separated DTO classes as well.

And finally, the `BookAppService` implementation is very simple:

```
public class BookAppService :
    CrudAppService<Book, BookDto, Guid, PagedAndSortedR
                        CreateUpdateBookDto, CreateUpda
    IBookAppService
{
    public BookAppService(IRepository<Book, Guid> repos
        : base(repository)
    {
    }
}
```

`CrudAppService` implements all methods declared in the `ICrudAppService` interface. You can then add your own custom methods or override and customize base methods.

> `CrudAppService` has different versions gets different number of generic arguments. Use the one suitable for you.

Share on :

In this document

Share on :

In this document

AbstractKeyCrudAppService

`CrudAppService` requires to have an Id property as the primary key of your entity. If you are using composite keys then you can not utilize it.

`AbstractKeyCrudAppService` implements the same `ICrudAppService` interface, but this time without making assumption about your primary key.

Example

Assume that you have a `District` entity with `CityId` and `Name` as a composite primary key. Using `AbstractKeyCrudAppService` requires to implement `DeleteByIdAsync` and `GetEntityByIdAsync` methods yourself:

```
public class DistrictAppService
    : AbstractKeyCrudAppService<District, DistrictDto,
{
    public DistrictAppService(IRepository<District> rep
        : base(repository)
    {
    }

    protected async override Task DeleteByIdAsync(Distr
    {
        await Repository.DeleteAsync(d => d.CityId == i
    }

    protected async override Task<District> GetEntityBy
    {
        return await AsyncQueryableExecuter.FirstOrDefa
            Repository.Where(d => d.CityId == id.CityId
        );
    }
}
```

This implementation requires you to create a class represents your composite key:

```
public class DistrictKey
{
    public Guid CityId { get; set; }

    public string Name { get; set; }
}
```

Authorization (for CRUD App Services)

There are two ways of authorizing the base application service methods;

1. You can set the policy properties (xxxPolicyName) in the constructor of your service. Example:

```
public class MyPeopleAppService : CrudAppService<Person
{
    public MyPeopleAppService(IRepository<Person, Guid>
        : base(repository)
    {
        GetPolicyName = "...";
        GetListPolicyName = "...";
        CreatePolicyName = "...";
        UpdatePolicyName = "...";
        DeletePolicyName = "...";
    }
}
```

`CreatePolicyName` is checked by the `CreateAsync` method and so on... You should specify a policy (permission) name defined in your application.

2. You can override the check methods (CheckXxxPolicyAsync) in your service. Example:

```
public class MyPeopleAppService : CrudAppService<Person
{
    public MyPeopleAppService(IRepository<Person, Guid>
        : base(repository)
    {
    }

    protected async override Task CheckDeletePolicyAsyn
    {
        await AuthorizationService.CheckAsync("...");
    }
}
```

You can perform any logic in the `CheckDeletePolicyAsync` method. It is expected to throw an `AbpAuthorizationException` in any unauthorized case, like `AuthorizationService.CheckAsync` already does.

Base Properties & Methods

CRUD application service base class provides many useful base methods that **you can override** to customize it based on your requirements.

CRUD Methods

These are the essential CRUD methods. You can override any of them to completely customize the operation. Here, the definitions of the methods:

```
Task<TGetOutputDto> GetAsync(TKey id);
Task<PagedResultDto<TGetListOutputDto>> GetListAsync(TG
Task<TGetOutputDto> CreateAsync(TCreateInput input);
Task<TGetOutputDto> UpdateAsync(TKey id, TUpdateInput i
Task DeleteAsync(TKey id);
```

Share on :

In this document

Share on :

In this document

Querying

These methods are low level methods those can be control how to query entities from the database.

- `CreateFilteredQuery` can be overridden to create an `IQueryable<TEntity>` that is filtered by the given input. If your `TGetListInput` class contains any filter, it is proper to override this method and filter the query. It returns the (unfiltered) repository (which is already `IQueryable<TEntity>`) by default.
- `ApplyPaging` is used to make paging on the query. If your `TGetListInput` already implements `IPagedResultRequest` , you don't need to override this since the ABP Framework automatically understands it and performs the paging.
- `ApplySorting` is used to sort (order by...) the query. If your `TGetListInput` already implements the `ISortedResultRequest` , ABP Framework automatically sorts the query. If not, it fallbacks to the `ApplyDefaultSorting` which tries to sort by creating time, if your entity implements the standard `IHasCreationTime` interface.
- `GetEntityByIdAsync` is used to get an entity by id, which calls `Repository.GetAsync(id)` by default.
- `DeleteByIdAsync` is used to delete an entity by id, which calls `Repository.DeleteAsync(id)` by default.

Object to Object Mapping

These methods are used to convert Entities to DTOs and vice verse. They uses the IObjectMapper by default.

- `MapToGetOutputDtoAsync` is used to map the entity to the DTO returned from the `GetAsync` , `CreateAsync` and `UpdateAsync` methods. Alternatively, you can override the `MapToGetOutputDto` if you don't need to perform any async operation.
- `MapToGetListOutputDtosAsync` is used to map a list of entities to a list of DTOs returned from the `GetListAsync` method. It uses the `MapToGetListOutputDtoAsync` to map each entity in the list. You can override one of them based on your case. Alternatively, you can override the `MapToGetListOutputDto` if you don't need to perform any async operation.
- `MapToEntityAsync` method has two overloads;
  - `MapToEntityAsync(TCreateInput)` is used to create an entity from `TCreateInput` .
  - `MapToEntityAsync(TUpdateInput, TEntity)` is used to update an existing entity from `TUpdateInput` .

Miscellaneous

Working with Streams

`Stream` object itself is not serializable. So, you may have problems if you directly use `Stream` as the parameter or the return value for your application service. ABP Framework provides a special type, `IRemoteStreamContent` to be used to get or return streams in the application services.

Example: Application Service Interface that can be used to get and return streams

```
using System;
using System.Threading.Tasks;
using Volo.Abp.Application.Services;
using Volo.Abp.Content;

namespace MyProject.Test
{
    public interface ITestAppService : IApplicationServ
    {
        Task Upload(Guid id, IRemoteStreamContent strea
        Task<IRemoteStreamContent> Download(Guid id);
    }
}
```

Example: Application Service Implementation that can be used to get and return streams

```
using System;
using System.IO;
using System.Threading.Tasks;
using Volo.Abp;
using Volo.Abp.Application.Services;
using Volo.Abp.Content;

namespace MyProject.Test
{
    public class TestAppService : ApplicationService, I
    {
        public Task<IRemoteStreamContent> Download(Guid
        {
            var fs = new FileStream("C:\\Temp\\" + id +
            return Task.FromResult(
                (IRemoteStreamContent) new RemoteStream
                    ContentType = "application/octet-st
                }
            );
        }

        public async Task Upload(Guid id, IRemoteStream
        {
            using (var fs = new FileStream("C:\\Temp\\"
            {
                await streamContent.GetStream().CopyToA
                await fs.FlushAsync();
            }
        }
    }
}
```

`IRemoteStreamContent` is compatible with the Auto API Controller and Dynamic C# HTTP Proxy systems.

Lifetime

Lifetime of application services are transient and they are automatically registered to the dependency injection system.

Share on :

In this document

4.1 (latest) English

Filter topics

Share on :

In this document