Articles Tutorials

Filter...

latest (v7.1)

In this document

Edit on GitHub

Application Services are used to expose domain logic to the presentation layer. An Application Service is called from the presentation layer using a DTO (Data Transfer Object) as a parameter. It also uses domain objects to perform some specific business logic and returns a DTO back to the presentation layer. Thus, the presentation layer is completely isolated from Domain layer.

In an ideally layered application, the presentation layer never directly works with domain objects.

IApplicationService Interface

In ASP.NET Boilerplate, an application service **should** implement the **IApplicationService** interface. It's good to create an **interface** for each Application Service.

First, let's define an interface for an application service:

Copy

```
public interface IPersonAppService : IApplicationService
{
    void CreatePerson(CreatePersonInput input);
}
```

IPersonAppService has only one method. It's used by the presentation layer to create a new person. **CreatePersonInput** is a DTO object as shown below:

Copy

```
public class CreatePersonInput
{
    [Required]
    public string Name { get; set; }

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

Now we can implement the IPersonAppService:

Copy

```
public class PersonAppService : IPersonAppService
{
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public void CreatePerson(CreatePersonInput input)
    {
        var person = _personRepository.FirstOrDefault(p => p.EmailAddress ==
input.EmailAddress);
        if (person != null)
        {
            throw new UserFriendlyException("There is already a person with given email
address");
        }

        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
    }
}
```

There are some important points to consider here:

- PersonAppService uses IRepository<Person> to perform database operations. It uses a **constructor injection** pattern, and thereby uses dependency injection.
- PersonAppService implements **IApplicationService** (since IPersonAppService extends IApplicationService). It's automatically registered to Dependency Injection system by ASP.NET Boilerplate and can be injected and used by other classes. The naming convention is important here. See the dependency injection document for more info.
- The **CreatePerson** method gets **CreatePersonInput**. It's an **input DTO** and automatically validated by ASP.NET Boilerplate. See the DTO and validation documents for details.

ApplicationService Class

An application service should implement the IApplicationService interface as declared above. **Optionally**, it can be derived from the **ApplicationService** base class. Thus, IApplicationService is inherently implemented.

The ApplicationService class has some basic functionality that makes it easy to do **logging, localization** and so on... It's recommended that you create a special base class for your application services that extends the ApplicationService class. This way, you can add some common functionality for all your application services. A sample application service class is shown below:

Copy

```
public class TaskAppService : ApplicationService, ITaskAppService
{
    public TaskAppService()
    {
        LocalizationSourceName = "SimpleTaskSystem";
    }

    public void CreateTask(CreateTaskInput input)
    {
        //Write some logs (Logger is defined in ApplicationService class)
        Logger.Info("Creating a new task with description: " + input.Description);

        //Get a localized text (L is a shortcut for LocalizationHelper.GetString(...),
defined in ApplicationService class)
        var text = L("SampleLocalizableTextKey");

        //TODO: Add new task to database...
    }
}
```

You can have a base class which defines **LocalizationSourceName** in it's constructor. This way you do not repeat it for all service classes. See the logging and localization documents for more information on this topic.

CrudAppService and AsyncCrudAppService Classes

If you need to create an application service that will have **Create, Update, Delete, Get, GetAll** methods for a **specific entity**, you can easily inherit from the **CrudAppService** class. You could also use the **AsyncCrudAppService** class to create async methods. The CrudAppService base class is **generic**, which gets the related **Entity** and **DTO** types as generic arguments. This is also **extensible**, allowing you to override functionality when you need to customize it.

Simple CRUD Application Service Example

Assume that we have a Task entity defined below:

Copy

```
public class Task : Entity, IHasCreationTime
{
    public string Title { get; set; }

    public string Description { get; set; }

    public DateTime CreationTime { get; set; }

    public TaskState State { get; set; }

    public Person AssignedPerson { get; set; }
    public Guid? AssignedPersonId { get; set; }

    public Task()
    {
        CreationTime = Clock.Now;
        State = TaskState.Open;
    }
}
```

And we created a DTO for this entity:

Copy

```
[AutoMap(typeof(Task))]
public class TaskDto : EntityDto, IHasCreationTime
{
    public string Title { get; set; }

    public string Description { get; set; }

    public DateTime CreationTime { get; set; }

    public TaskState State { get; set; }

    public Guid? AssignedPersonId { get; set; }

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

The AutoMap attribute creates a mapping configuration between the entity and dto. Now, we can create an application service as shown below:

Copy

```
public class TaskAppService : AsyncCrudAppService<Task, TaskDto>
{
    public TaskAppService(IRepository<Task> repository)
        : base(repository)
    {

    }
}
```

We injected the repository and passed it to the base class (We could inherit from CrudAppService if we want to create sync methods instead of async methods).

That's all! TaskAppService now has simple CRUD methods!

If you want to define an interface for the application service, you can create your interface like this:

Copy

```
public interface ITaskAppService : IAsyncCrudAppService<TaskDto>
{

}
```

Notice that **IAsyncCrudAppService** does not get the entity (Task) as a generic argument. This is because the entity is related to the implementation and should not be included in a public interface.

We can now implement the ITaskAppService interface for the TaskAppService class:

Copy

```
public class TaskAppService : AsyncCrudAppService<Task, TaskDto>, ITaskAppService
{
    public TaskAppService(IRepository<Task> repository)
        : base(repository)
    {

    }
}
```

Customize CRUD Application Services

Getting a List

A Crud application service gets **PagedAndSortedResultRequestDto** as an argument for the **GetAll** method as default, which provides optional sorting and paging parameters. You may also want to add other parameters for the GetAll method. For example, you may want to add some **custom filters**. In this case, you can create a DTO for the GetAll method. Example:

Copy

```
public class GetAllTasksInput : PagedAndSortedResultRequestDto
{
    public TaskState? State { get; set; }
}
```

Here we inherit from **PagedAndSortedResultRequestInput**. This is **not required**, but if you want, you can use the paging & sorting parameters from the base class. We also added an **optional State** property to filter tasks by state. With this, we change the TaskAppService class in order to apply the **custom filter**:

Copy

```
public class TaskAppService : AsyncCrudAppService<Task, TaskDto, int, GetAllTasksInput>
{
    public TaskAppService(IRepository<Task> repository)
        : base(repository)
    {

    }

    protected override IQueryable<Task> CreateFilteredQuery(GetAllTasksInput input)
    {
        return base.CreateFilteredQuery(input)
            .WhereIf(input.State.HasValue, t => t.State == input.State.Value);
    }
}
```

First, we added **GetAllTasksInput** as a 4th generic parameter to the AsyncCrudAppService class (3rd one is PK type of the entity). Then we override the **CreateFilteredQuery** method to apply custom filters. This method is an extension point for the AsyncCrudAppService class. Note that WhereIf is an extension method of ABP to simplify conditional filtering. What we're doing here is simply filtering an IQueryable.

Note: If you created an application service interface, you need to add the same generic arguments to that interface, too!

Create and Update

Notice that we are using same DTO (TaskDto) for getting, **creating** and **updating** tasks which may not be good for real life applications, so we may want to **customize the create and update DTOs**.

Let's start by creating a **CreateTaskInput** class:

Copy

```
[AutoMapTo(typeof(Task))]
public class CreateTaskInput
{
    [Required]
    [StringLength(Task.MaxTitleLength)]
    public string Title { get; set; }

    [StringLength(Task.MaxDescriptionLength)]
    public string Description { get; set; }

    public Guid? AssignedPersonId { get; set; }
}
```

In addition to this, create an **UpdateTaskInput** DTO:

Copy

```
[AutoMapTo(typeof(Task))]
public class UpdateTaskInput : CreateTaskInput, IEntityDto
{
    public int Id { get; set; }

    public TaskState State { get; set; }
}
```

Here we inherit from **CreateTaskInput** to include all properties for the Update operation (you may want something different). Implementing **IEntity** (or IEntity<PrimaryKey> for a different PK than int) is **required** here, because we need to know which entity is being updated. Lastly, we added an additional property, **State**, which is not in CreateTaskInput.

We can now use these DTO classes as generic arguments for the AsyncCrudAppService class:

Copy

```
public class TaskAppService : AsyncCrudAppService<Task, TaskDto, int, GetAllTasksInput,
CreateTaskInput, UpdateTaskInput>
{
    public TaskAppService(IRepository<Task> repository)
        : base(repository)
    {

    }

    protected override IQueryable<Task> CreateFilteredQuery(GetAllTasksInput input)
    {
        return base.CreateFilteredQuery(input)
            .WhereIf(input.State.HasValue, t => t.State == input.State.Value);
    }
}
```

No need for any additional code changes!

Other Method Arguments

AsyncCrudAppService can get more generic arguments if you want to customize input DTOs for **Get** and **Delete** methods. All methods of the base class are virtual, so you can override them to customize the behaviour.

CRUD Permissions

Do you need to authorize your CRUD methods? If so, there are pre-defined permission properties you can set: GetPermissionName, GetAllPermissionName, CreatePermissionName, UpdatePermissionName and DeletePermissionName. The base CRUD class automatically checks permissions if you set them.

Here, you can set it in the constructor:

Copy

```
public class TaskAppService : AsyncCrudAppService<Task, TaskDto>
{
    public TaskAppService(IRepository<Task> repository)
        : base(repository)
    {
        CreatePermissionName = "MyTaskCreationPermission";
    }
}
```

Alternatively, you can override the appropriate permission checker methods to manually check permissions: CheckGetPermission(), CheckGetAllPermission(), CheckCreatePermission(), CheckUpdatePermission(), CheckDeletePermission(). By default, they all call the CheckPermission(...) method with the related permission name.
Simply, this calls the IPermissionChecker.Authorize(...) method.

Unit of Work

An application service method is a **unit of work** by default in ASP.NET Boilerplate. Thus, the application service methods are transactional and automatically save all database changes when each method ends.

See the unit of work documentation for more information.

Lifetime of an Application Service

All application service instances are **Transient**. This means they are instantiated each time.

See the Dependency Injection documentation for more information.