Search in documents Contributors Edit Last edit: 9/19/2020 Share on :

In this document

This document has multiple versions. Select the options best fit for you.

UI MVC / Ra Database Er

Web Application Development Tutorial - Part 8: Authors: Application Layer

About This Tutorial

In this tutorial series, you will build an ABP based web application named `Acme.BookStore`. This application is used to manage a list of books and their authors. It is developed using the following technologies:

- **Entity Framework Core** as the ORM provider.
- **MVC / Razor Pages** as the UI Framework.

This tutorial is organized as the following parts;

- Part 1: Creating the server side
- Part 2: The book list page
- Part 3: Creating, updating and deleting books
- Part 4: Integration tests
- Part 5: Authorization
- Part 6: Authors: Domain layer
- Part 7: Authors: Database Integration
- **Part 8: Author: Application Layer (this part)**
- Part 9: Authors: User Interface
- Part 10: Book to Author Relation

Download the Source Code

This tutorial has multiple versions based on your **UI** and **Database** preferences. We've prepared a few combinations of the source code to be downloaded:

- MVC (Razor Pages) UI with EF Core
- Blazor UI with EF Core
- Angular UI with MongoDB

Introduction

This part explains to create an application layer for the `Author` entity created before.

IAuthorAppService

We will first create the application service interface and the related DTOs. Create a new interface, named `IAuthorAppService`, in the `Authors` namespace (folder) of the `Acme.BookStore.Application.Contracts` project:

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

namespace Acme.BookStore.Authors
{
    public interface IAuthorAppService : IApplicationSe
    {
        Task<AuthorDto> GetAsync(Guid id);

        Task<PagedResultDto<AuthorDto>> GetListAsync(Ge

        Task<AuthorDto> CreateAsync(CreateAuthorDto inp

        Task UpdateAsync(Guid id, UpdateAuthorDto input

        Task DeleteAsync(Guid id);
    }
}
```

- `IApplicationService` is a conventional interface that is inherited by all the application services, so the ABP Framework can identify the service.
- Defined standard methods to perform CRUD operations on the `Author` entity.
- `PagedResultDto` is a pre-defined DTO class in the ABP Framework. It has an `Items` collection and a `TotalCount` property to return a paged result.
- Preferred to return an `AuthorDto` (for the newly created author) from the `CreateAsync` method, while it is not used by this application - just to show a different usage.

This interface is using the DTOs defined below (create them for your project).

AuthorDto

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

namespace Acme.BookStore.Authors
{
    public class AuthorDto : EntityDto<Guid>
    {
        public string Name { get; set; }

        public DateTime BirthDate { get; set; }

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

- `EntityDto<T>` simply has an `Id` property with the given generic argument. You could create an `Id` property yourself instead of inheriting the `EntityDto<T>`.

Share on :

In this document

Share on :

In this document

GetAuthorListDto

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

namespace Acme.BookStore.Authors
{
    public class GetAuthorListDto : PagedAndSortedResul
    {
        public string Filter { get; set; }
    }
}
```

- `Filter` is used to search authors. It can be `null` (or empty string) to get all the authors.
- `PagedAndSortedResultRequestDto` has the standard paging and sorting properties: `int MaxResultCount`, `int SkipCount` and `string Sorting`.

> ABP Framework has such base DTO classes to simplify and standardize your DTOs. See the DTO documentation for all.

CreateAuthorDto

```
using System;
using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore.Authors
{
    public class CreateAuthorDto
    {
        [Required]
        [StringLength(AuthorConsts.MaxNameLength)]
        public string Name { get; set; }

        [Required]
        public DateTime BirthDate { get; set; }

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

Data annotation attributes can be used to validate the DTO. See the validation document for details.

UpdateAuthorDto

```
using System;
using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore.Authors
{
    public class UpdateAuthorDto
    {
        [Required]
        [StringLength(AuthorConsts.MaxNameLength)]
        public string Name { get; set; }

        [Required]
        public DateTime BirthDate { get; set; }

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

> We could share (re-use) the same DTO among the create and the update operations. While you can do it, we prefer to create different DTOs for these operations since we see they generally be different by the time. So, code duplication is reasonable here compared to a tightly coupled design.

AuthorAppService

It is time to implement the `IAuthorAppService` interface. Create a new class, named `AuthorAppService` in the `Authors` namespace (folder) of the `Acme.BookStore.Application` project:

Share on :

In this document

Share on :

In this document

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Acme.BookStore.Permissions;
using Microsoft.AspNetCore.Authorization;
using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Authors
{
    [Authorize(BookStorePermissions.Authors.Default)]
    public class AuthorAppService : BookStoreAppService
    {
        private readonly IAuthorRepository _authorRepos
        private readonly AuthorManager _authorManager;

        public AuthorAppService(
            IAuthorRepository authorRepository,
            AuthorManager authorManager)
        {
            _authorRepository = authorRepository;
            _authorManager = authorManager;
        }

        //...SERVICE METHODS WILL COME HERE...
    }
}
```

- `[Authorize(BookStorePermissions.Authors.Default)]` is a declarative way to check a permission (policy) to authorize the current user. See the authorization document for more. `BookStorePermissions` class will be updated below, don't worry for the compile error for now.
- Derived from the `BookStoreAppService`, which is a simple base class comes with the startup template. It is derived from the standard `ApplicationService` class.
- Implemented the `IAuthorAppService` which was defined above.
- Injected the `IAuthorRepository` and `AuthorManager` to use in the service methods.

Now, we will introduce the service methods one by one. Copy the explained method into the `AuthorAppService` class.

GetAsync

```
public async Task<AuthorDto> GetAsync(Guid id)
{
    var author = await _authorRepository.GetAsync(id);
    return ObjectMapper.Map<Author, AuthorDto>(author);
}
```

This method simply gets the `Author` entity by its `Id`, converts to the `AuthorDto` using the object to object mapper. This requires to configure the AutoMapper, which will be explained later.

GetListAsync

Share on :

In this document

```
public async Task<PagedResultDto<AuthorDto>> GetListAsy
{
    if (input.Sorting.IsNullOrWhiteSpace())
    {
        input.Sorting = nameof(Author.Name);
    }

    var authors = await _authorRepository.GetListAsync(
        input.SkipCount,
        input.MaxResultCount,
        input.Sorting,
        input.Filter
    );

    var totalCount = await AsyncExecuter.CountAsync(
        _authorRepository.WhereIf(
            !input.Filter.IsNullOrWhiteSpace(),
            author => author.Name.Contains(input.Filter
        )
    );

    return new PagedResultDto<AuthorDto>(
        totalCount,
        ObjectMapper.Map<List<Author>, List<AuthorDto>>
    );
}
```

- Default sorting is "by author name" which is done in the beginning of the method in case of it wasn't sent by the client.
- Used the `IAuthorRepository.GetListAsync` to get a paged, sorted and filtered list of authors from the database. We had implemented it in the previous part of this tutorial. Again, it actually was not needed to create such a method since we could directly query over the repository, but wanted to demonstrate how to create custom repository methods.
- Directly queried from the `AuthorRepository` while getting the count of the authors. We preferred to use the `AsyncExecuter` service which allows us to perform async queries without depending on the EF Core. However, you could depend on the EF Core package and directly use the `_authorRepository.WhereIf(...).ToListAsync()` method. See the repository document to read the alternative approaches and the discussion.
- Finally, returning a paged result by mapping the list of `Author` s to a list of `AuthorDto` s.

CreateAsync

Share on :

In this document

```
[Authorize(BookStorePermissions.Authors.Create)]
public async Task<AuthorDto> CreateAsync(CreateAuthorDt
{
    var author = await _authorManager.CreateAsync(
        input.Name,
        input.BirthDate,
        input.ShortBio
    );

    await _authorRepository.InsertAsync(author);

    return ObjectMapper.Map<Author, AuthorDto>(author);
}
```

- `CreateAsync` requires the `BookStorePermissions.Authors.Create` permission (in addition to the `BookStorePermissions.Authors.Default` declared for the `AuthorAppService` class).
- Used the `AuthorManeger` (domain service) to create a new author.
- Used the `IAuthorRepository.InsertAsync` to insert the new author to the database.
- Used the `ObjectMapper` to return an `AuthorDto` representing the newly created author.

> **DDD tip**: Some developers may find useful to insert the new entity inside the `_authorManager.CreateAsync`. We think it is a better design to leave it to the application layer since it better knows when to insert it to the database (maybe it requires additional works on the entity before insert, which would require to an additional update if we perform the insert in the domain service). However, it is completely up to you.

UpdateAsync

```
[Authorize(BookStorePermissions.Authors.Edit)]
public async Task UpdateAsync(Guid id, UpdateAuthorDto
{
    var author = await _authorRepository.GetAsync(id);

    if (author.Name != input.Name)
    {
        await _authorManager.ChangeNameAsync(author, in
    }

    author.BirthDate = input.BirthDate;
    author.ShortBio = input.ShortBio;

    await _authorRepository.UpdateAsync(author);
}
```

- `UpdateAsync` requires the additional `BookStorePermissions.Authors.Edit` permission.
- Used the `IAuthorRepository.GetAsync` to get the author entity from the database. `GetAsync` throws `EntityNotFoundException` if there is no author with the given id, which results a `404` HTTP status code

in a web application. It is a good practice to always bring the entity on an update operation.

- Used the `AuthorManager.ChangeNameAsync` (domain service method) to change the author name if it was requested to change by the client.
- Directly updated the `BirthDate` and `ShortBio` since there is not any business rule to change these properties, they accept any value.
- Finally, called the `IAuthorRepository.UpdateAsync` method to update the entity on the database.

> **EF Core Tip**: Entity Framework Core has a **change tracking** system and **automatically saves** any change to an entity at the end of the unit of work (You can simply think that the ABP Framework automatically calls `SaveChanges` at the end of the method). So, it will work as expected even if you don't call the `_authorRepository.UpdateAsync(...)` in the end of the method. If you don't consider to change the EF Core later, you can just remove this line.

DeleteAsync

```
[Authorize(BookStorePermissions.Authors.Delete)]
public async Task DeleteAsync(Guid id)
{
    await _authorRepository.DeleteAsync(id);
}
```

- `DeleteAsync` requires the additional `BookStorePermissions.Authors.Delete` permission.
- It simply uses the `DeleteAsync` method of the repository.

Permission Definitions

You can't compile the code since it is expecting some constants declared in the `BookStorePermissions` class.

Open the `BookStorePermissions` class inside the `Acme.BookStore.Application.Contracts` project (in the `Permissions` folder) and change the content as shown below:

```
namespace Acme.BookStore.Permissions
{
    public static class BookStorePermissions
    {
        public const string GroupName = "BookStore";

        public static class Books
        {
            public const string Default = GroupName + "
            public const string Create = Default + ".Cr
            public const string Edit = Default + ".Edit
            public const string Delete = Default + ".De
        }

        // *** ADDED a NEW NESTED CLASS ***
        public static class Authors
        {
            public const string Default = GroupName + "
            public const string Create = Default + ".Cr
            public const string Edit = Default + ".Edit
            public const string Delete = Default + ".De
        }
    }
}
```

Then open the `BookStorePermissionDefinitionProvider` in the same project and add the following lines at the end of the `Define` method:

```
var authorsPermission = bookStoreGroup.AddPermission(
    BookStorePermissions.Authors.Default, L("Permission

authorsPermission.AddChild(
    BookStorePermissions.Authors.Create, L("Permission:

authorsPermission.AddChild(
    BookStorePermissions.Authors.Edit, L("Permission:Au

authorsPermission.AddChild(
    BookStorePermissions.Authors.Delete, L("Permission:
```

Finally, add the following entries to the `Localization/BookStore/en.json` inside the `Acme.BookStore.Domain.Shared` project, to localize the permission names:

```
"Permission:Authors": "Author Management",
"Permission:Authors.Create": "Creating new authors",
"Permission:Authors.Edit": "Editing the authors",
"Permission:Authors.Delete": "Deleting the authors"
```

Object to Object Mapping

Share on :

In this document

`AuthorAppService` is using the `ObjectMapper` to convert the `Author` objects to `AuthorDto` objects. So, we need to define this mapping in the AutoMapper configuration.

Open the `BookStoreApplicationAutoMapperProfile` class inside the `Acme.BookStore.Application` project and add the following line to the constructor:

```
CreateMap<Author, AuthorDto>();
```

Data Seeder

As just done for the books before, it would be good to have some initial author entities in the database. This will be good while running the application first time, but also it is very useful for the automated tests.

Open the `BookStoreDataSeederContributor` in the `Acme.BookStore.Domain` project and change the file content with the code below:

Share on :

In this document

```
using System;
using System.Threading.Tasks;
using Acme.BookStore.Authors;
using Acme.BookStore.Books;
using Volo.Abp.Data;
using Volo.Abp.DependencyInjection;
using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore
{
    public class BookStoreDataSeederContributor
        : IDataSeedContributor, ITransientDependency
    {
        private readonly IRepository<Book, Guid> _bookR
        private readonly IAuthorRepository _authorRepos
        private readonly AuthorManager _authorManager;

        public BookStoreDataSeederContributor(
            IRepository<Book, Guid> bookRepository,
            IAuthorRepository authorRepository,
            AuthorManager authorManager)
        {
            _bookRepository = bookRepository;
            _authorRepository = authorRepository;
            _authorManager = authorManager;
        }

        public async Task SeedAsync(DataSeedContext con
        {
            if (await _bookRepository.GetCountAsync() <
            {
                await _bookRepository.InsertAsync(
                    new Book
                    {
                        Name = "1984",
                        Type = BookType.Dystopia,
                        PublishDate = new DateTime(1949
                        Price = 19.84f
                    },
                    autoSave: true
                );

                await _bookRepository.InsertAsync(
                    new Book
                    {
                        Name = "The Hitchhiker's Guide
                        Type = BookType.ScienceFiction,
                        PublishDate = new DateTime(1995
                        Price = 42.0f
                    },
                    autoSave: true
                );
            }

            // ADDED SEED DATA FOR AUTHORS

            if (await _authorRepository.GetCountAsync()
            {
                await _authorRepository.InsertAsync(
                    await _authorManager.CreateAsync(
                        "George Orwell",
                    new DateTime(1903, 06, 25),
                    "Orwell produced literary criti
                )
            );

            await _authorRepository.InsertAsync(
                await _authorManager.CreateAsync(
                    "Douglas Adams",
                    new DateTime(1952, 03, 11),
                    "Douglas Adams was an English a
                )
            );
        }
    }
}
}
```

You can now run the `.DbMigrator` console application to **migrate** the **database schema** and **seed** the initial data.

Testing the Author Application Service

Finally, we can write some tests for the `IAuthorAppService`. Add a new class, named `AuthorAppService_Tests` in the `Authors` namespace (folder) of the `Acme.BookStore.Application.Tests` project:

Share on :

In this document

```
using System;
using System.Threading.Tasks;
using Shouldly;
using Xunit;

namespace Acme.BookStore.Authors
{
    public class AuthorAppService_Tests : BookStoreAppl
    {
        private readonly IAuthorAppService _authorAppSe

        public AuthorAppService_Tests()
        {
            _authorAppService = GetRequiredService<IAut
        }

        [Fact]
        public async Task Should_Get_All_Authors_Withou
        {
            var result = await _authorAppService.GetLis

            result.TotalCount.ShouldBeGreaterThanOrEqua
            result.Items.ShouldContain(author => author
            result.Items.ShouldContain(author => author
        }

        [Fact]
        public async Task Should_Get_Filtered_Authors()
        {
            var result = await _authorAppService.GetLis
                new GetAuthorListDto {Filter = "George"

            result.TotalCount.ShouldBeGreaterThanOrEqua
            result.Items.ShouldContain(author => author
            result.Items.ShouldNotContain(author => aut
        }

        [Fact]
        public async Task Should_Create_A_New_Author()
        {
            var authorDto = await _authorAppService.Cre
                new CreateAuthorDto
                {
                    Name = "Edward Bellamy",
                    BirthDate = new DateTime(1850, 05,
                    ShortBio = "Edward Bellamy was an A
                }
            );

            authorDto.Id.ShouldNotBe(Guid.Empty);
            authorDto.Name.ShouldBe("Edward Bellamy");
        }

        [Fact]
        public async Task Should_Not_Allow_To_Create_Du
        {
            await Assert.ThrowsAsync<AuthorAlreadyExist
            {
                await _authorAppService.CreateAsync(
                    new CreateAuthorDto
                    {
```

Share on :

In this document

```
                        Name = "Douglas Adams",
                        BirthDate = DateTime.Now,
                        ShortBio = "..."
                    }
                );
            });
        }

        //TODO: Test other methods...
    }
}
```

Created some tests for the application service methods, which should be clear to understand.

The Next Part

See the next part of this tutorial.

Share on :

In this document