0ABP.IO Framework

1.UI Framework Options

MVC Razor Pages | ANGULAR | Blazor

We Will Use Blazor as a UI.

2. Database Provider Options

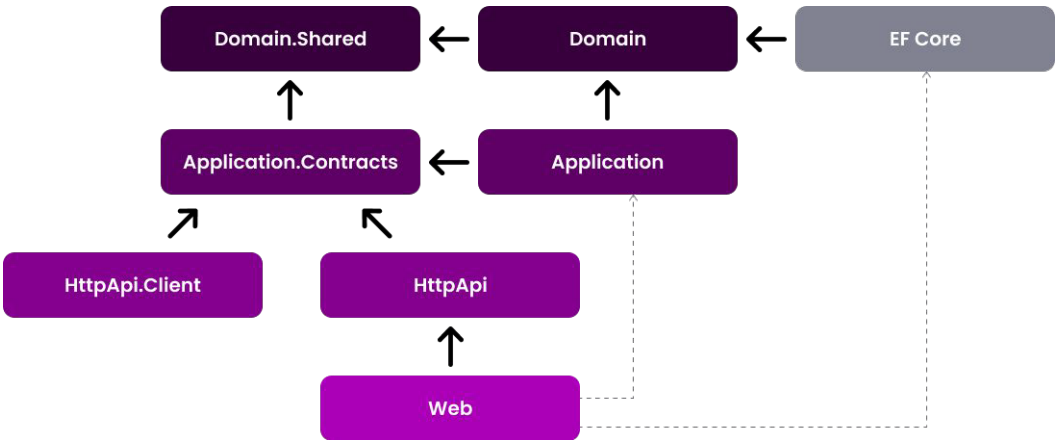
Entity Framework Core | mongoDB | Dapper

3. Architecture

ABP offers a complete, modular and layered software architecture based on Domain Driven Design principles and patterns. It also provides the necessary infrastructure to implement this architecture.

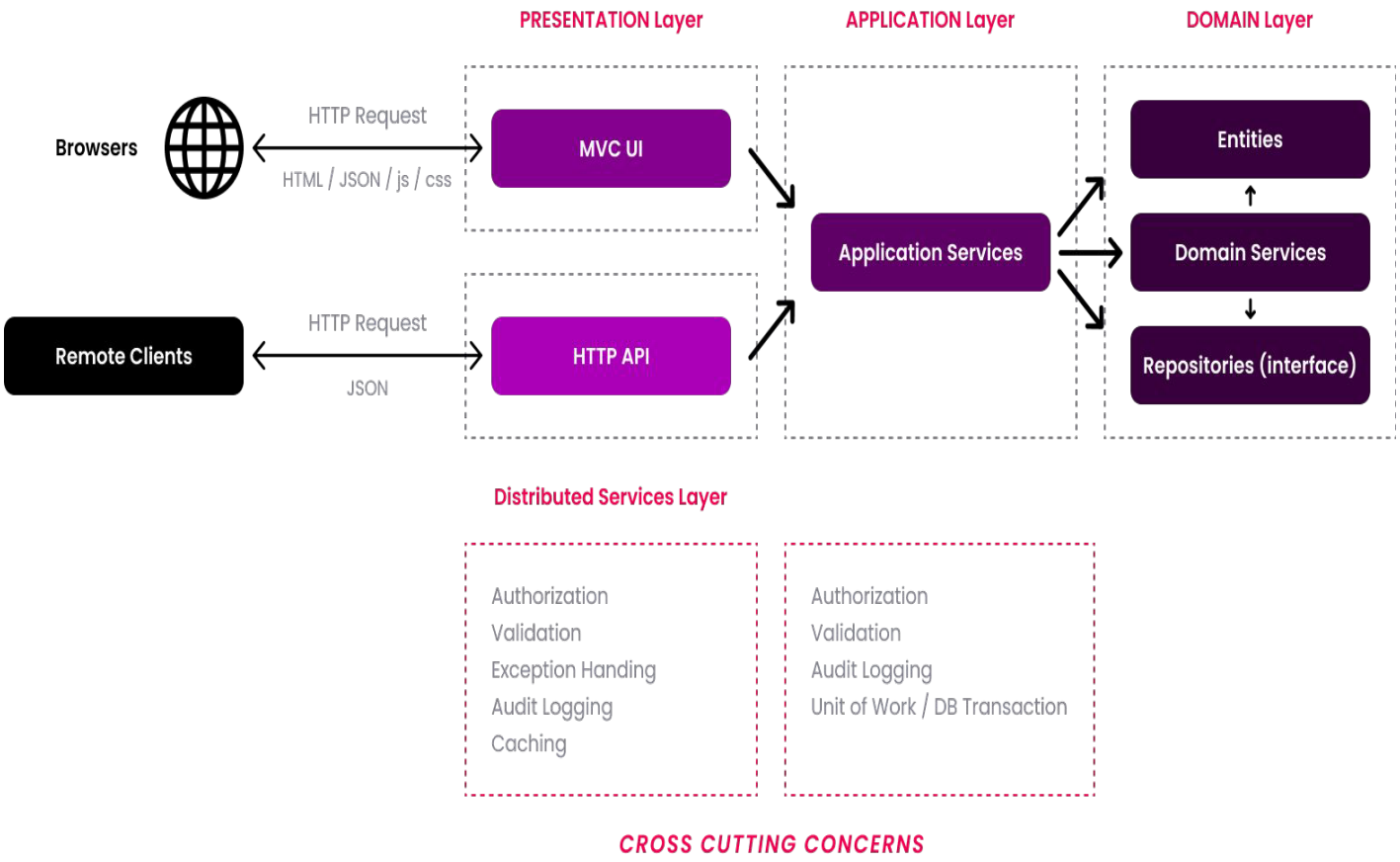
- See the Modularity document to understand the module system.
- Implementing Domain Driven Design book is an ultimate guide for who want to understand and implement the DDD with the ABP Framework.
- Microservice Architecture document explains how ABP helps to create a microservice solution.
- Multi-Tenancy document introduces multi-tenancy and explores the ABP multi-tenancy infrastructure.

Domain Driven Design:



- Domain.Shared is the project that all other projects directly or indirectly depend on. So, all the types in this project are available to all projects.
- Domain only depends on the Domain.Shared because it is already a (shared) part of the domain. For example, an IssueType enum in the Domain.Shared can be used by an Issue **entity** in the Domain project.

- Application.Contracts depends on the Domain.Shared. In this way, you can reuse these types in the DTOs. For example, the same IssueType enum in the Domain.Shared can be used by a CreateIssueDto as a property.
- Application depends on the Application.Contracts since it implements the Application Service interfaces and uses the **DTOs** inside it. It also depends on the Domain since the Application Services are implemented using the Domain Objects defined inside it.
- EntityFrameworkCore depends on the Domain since it maps the Domain Objects (entities and value types) to database tables (as it is an ORM) and implements the repository interfaces defined in the Domain.
- HttpApi depends on the Application.Contracts since the Controllers inside it inject and use the Application Service interfaces as explained before.
- HttpApi.Client depends on the Application.Contracts since it can consume the Application Services as explained before.
- Web depends on the HttpApi since it serves the HTTP APIs defined inside it. Also, in this way, it indirectly depends on the Application.Contracts project to consume the Application Services in the Pages/Components.



1. Single-Layer Solution

Step-1: Creating a New Solution

I. dotnet tool install -g Volo.Abp.Cli

II. abp new TodoApp -t app-nolayers -u blazor-server

Step-2: Create the Database

I. dotnet run --migrate-database

II. abp install-libs (for Client-Side Packages)

III. abp bundle(For Bundling and Minification and it is automatically run)

Step-3: Run the Application

I. dotnet run

II. Default Admin Password---> admin && 1q2w3E*

Step-4: Defining the Entity

1. create a new TodoItem class under the Entities folder of the project.

```
using Volo.Abp.Domain.Entities;

namespace TodoAppEntities;

public class TodoItem : BasicAggregateRoot<Guid>{

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

Note:- BasicAggregateRoot is the simplest base class to create root entities, and Guid is the primary key (Id) of the entity here.

Step-5: Database Integration

1. Mapping Configuration:

Open the TodoAppDbContext class (in the Data folder) and add a new DbSet property to this class:

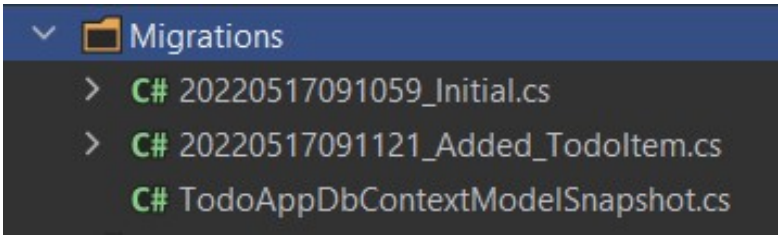
```
        public DbSet<TodoItem> TodoItems { get; set; }

        protected override void OnModelCreating(ModelBuilder builder){

        base.OnModelCreating(builder);
```

```
        /* Include modules to your migration db context */



        builder.ConfigurePermissionManagement();
...

/* Configure your own tables/entities inside here */
builder.Entity<TodoItem>(b =>
{
    b.ToTable("TodoItems");
});}
```

2. Code First Migrations:

I. Open a command-line terminal in the root directory of your project (in the same folder of the .csproj file) and type the following command:

```
dotnet ef migrations add Added_TodoItem
```



Ii. Then, you can apply changes to the database using the following command, in the same command-line terminal:

```
dotnet ef database update
```

Step-6: Creating the Application Service

An application service is used to perform the use cases of the application. We need to perform the following use cases in this application:

- Get the list of the todo items
- Create a new todo item
- Delete an existing todo item

1. Creating the Data Transfer Object (DTO):-

Application services typically get and return DTOs (Data Transfer Objects) instead of entities. So, create a new TodoItemDto class under the Services/Dtos folder:

```
namespace TodoApp.Services.Dtos;

public class TodoItemDto{

   public Guid Id { get; set; }

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

2.The Application Service Implementation;-

Create a TodoAppService class under the Services folder of your project, as shown below

```
using TodoApp.Entities;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;




namespace TodoApp.Services;

public class TodoAppService : ApplicationService{

   private readonly IRepository<TodoItem, Guid> _todoItemRepository;



   public TodoAppService(IRepository<TodoItem, Guid> todoItemRepository)

   {
```

```
    _todoItemRepository = todoItemRepository;

  }



  // TODO: Implement the methods here...}
```

:

This class injects IRepository<TodoItem, Guid>, which is the default repository for the TodoItem entity. We will use it to implement our use cases.

I. Getting the Todo Items

```
public async Task<List<TodoItemDto>> GetListAsync(){

  var items = await _todoItemRepository.GetListAsync();

  return items

    .Select(item => new TodoItemDto

    {

      Id = item.Id,

      Text = item.Text

    }).ToList();}
```

We are simply getting the TodoItem list from the repository, mapping them to the TodoItemDto objects and returning as the result.

Ii. Creating a New Todo Item

```
public async Task<TodoItemDto> CreateAsync(string text){

  var todoItem = await _todoItemRepository.InsertAsync(

    new TodoItem {Text = text}

  );



  return new TodoItemDto

  {

    Id = todoItem.Id,
```

```
    Text = todoItem.Text

  };}
```

The repository's InsertAsync method inserts the given TodoItem to the database and returns the same TodoItem object. It also sets the Id, so we can use it on the returning object. We are simply returning a TodoItemDto by creating from the new TodoItem entity.

Iii. Deleting a Todo Item

```
public async Task DeleteAsync(Guid id){

  await _todoItemRepository.DeleteAsync(id);}
```

Step-7: User Interface Implementation

1. Index.razor.cs

Open the Index.razor.cs file in the Pages folder and replace the content with the following code block:

```
using Microsoft.AspNetCore.Components;using TodoApp.Services;using TodoApp.Services.Dtos;



namespace TodoApp.Pages;

public partial class Index{

  [Inject]

  private TodoAppService TodoAppService { get; set; }



  private List<TodoItemDto> TodoItems { get; set; } = new List<TodoItemDto>();

  private string NewTodoText { get; set; }



  protected override async Task OnInitializedAsync()

  {
```

```
    TodoItems = await TodoAppService.GetListAsync();

     }

      private async Task Create()

  {

    var result = await TodoAppService.CreateAsync(NewTodoText);

    TodoItems.Add(result);

    NewTodoText = null;

  }



  private async Task Delete(TodoItemDto todoItem)

  {

    await TodoAppService.DeleteAsync(todoItem.Id);

    await Notify.Info("Deleted the todo item.");

    TodoItems.Remove(todoItem);

  }}
```

2. Index.razor

```
@page "/"

@inherits TodoAppComponentBase

<div class="container">

  <Card>

    <CardHeader>

      <CardTitle>

        TODO LIST

      </CardTitle>
```

```
    </CardHeader>

    <CardBody>

        <!-- FORM FOR NEW TODO ITEMS -->

        <form id="NewItemForm" @onsubmit:preventDefault @onsubmit="() => Create()" class="row row-cols-lg-auto g-3
align-items-center">

            <div class="col-12">

                <div class="input-group">

                    <input name="NewTodoText" type="text" @bind-value="@NewTodoText" class="form-control"
placeholder="enter text..." />

                </div>

            </div>

            <div class="col-12">

                <button type="submit" class="btn btn-primary">Submit</button>

            </div>

        </form>

        <!-- TODO ITEMS LIST -->

        <ul id="TodoList">

            @foreach (var todoItem in TodoItems)

            {

                <li data-id="@todoItem.Id">

                    <i class="far fa-trash-alt"

                       @onclick="() => Delete(todoItem)"></i>

                    @todoItem.Text

                </li>

            }

        </ul>
```

```
    </CardBody>

  </Card></div>
```

3. Index.razor.css

```
#TodoList{

   list-style: none;

   margin: 0;

   padding: 0;}

#TodoList li {

   padding: 5px;

   margin: 5px 0px;

   border: 1px solid #cccccc;

   background-color: #f5f5f5;}

#TodoList li i{

   opacity: 0.5;}

#TodoList li i:hover{

   opacity: 1;

   color: #ff0000;

   cursor: pointer;}
```

2. ***Multi-Layer Solution***

```
abp new TodoApp -u blazor-server
```

Create the Database: -

If you are using Visual Studio, right click on the **TodoApp.DbMigrator project**, select Set as StartUp Project, then hit Ctrl+F5 to run it without debugging. It will create the initial database and seed the initial data.

> Some IDEs (e.g. Rider) may have problems for the first run since DbMigrator adds the initial migration and re-compiles the project. In this case, open a command-line terminal in the folder of the .DbMigrator project and execute the dotnet run command.

Step-1: Domain Layer(TodoApp.Domain)

This application has a single entity and we'll start by creating it. Create a new TodoItem class inside **the *TodoApp.Domain* project:**

```
using System;using Volo.Abp.Domain.Entities;

namespace TodoApp{

  public class TodoItem : BasicAggregateRoot<Guid>

  {

    public string Text { get; set; }

  }}
```

Step-2: Database Integration(TodoApp.*EntityFrameworkCore)*

Open the TodoAppDbContext class in the EntityFrameworkCore folder of the *TodoApp.EntityFrameworkCore* project and add a new DbSet property to this class:

```
public DbSet<TodoItem> TodoItems { get; set; }

protected override void OnModelCreating(ModelBuilder builder){

  base.OnModelCreating(builder);



  /* Include modules to your migration db context */
```

```
builder.ConfigurePermissionManagement();

...



/* Configure your own tables/entities inside here */

builder.Entity<TodoItem>(b =>

{

    b.ToTable("TodoItems");

});}
```

Note:- **We've mapped the TodoItem entity to the TodoItems table in the database.**

##Code First Migrations:

Open a command-line terminal in the directory of the ***TodoApp.EntityFrameworkCore*** **project** and type the following command:

```
dotnet ef migrations add Added_TodoItem

dotnet ef database update
```

If you are using Visual Studio, you may want to use the Add-Migration Added_TodoItem and Update-Database commands in the *Package Manager Console (PMC)*. In this case, **ensure that TodoApp.Blazor is the startup project and TodoApp.EntityFrameworkCore is the *Default Project* in PMC.**

Step-3:Application Layer (For Interface and DTOs use TodoApp.Application.Contracts , For Application Services use TodoApp.Application)

An Application Service is used to perform the use cases of the application. We need to perform the following use cases:

- Get the list of the todo items
- Create a new todo item
- Delete an existing todo item

I. Application Service Interface:

We can start by defining an interface for the application service. Create a new **ITodoAppService interface in the *TodoApp.Application.Contracts* project**, as shown below:

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

namespace TodoApp{

  public interface ITodoAppService : IApplicationService

  {

    Task<List<TodoItemDto>> GetListAsync();

    Task<TodoItemDto> CreateAsync(string text);

    Task DeleteAsync(Guid id);

  }}
```

Ii. Data Transfer Object(DTOs)

GetListAsync and CreateAsync methods return TodoItemDto. ApplicationService typically gets and returns DTOs (Data Transfer Objects) instead of entities. So, we should define the DTO class here. **Create a new TodoItemDto class inside the TodoApp.Application.Contracts project:**

```
using System;

namespace TodoApp{

  public class TodoItemDto

  {

    public Guid Id { get; set; }

    public string Text { get; set; }

  }}
```

Iii. Application Service Implementation

Create a TodoAppService class inside the *TodoApp.Application* project, as shown below:

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

namespace TodoApp{
```

```
public class TodoAppService : ApplicationService, ITodoAppService

{

    private readonly IRepository<TodoItem, Guid> _todoItemRepository;



    public TodoAppService(IRepository<TodoItem, Guid> todoItemRepository)

    {

        _todoItemRepository = todoItemRepository;

    }



    // TODO: Implement the methods here...

}}
```

This class inherits from the ApplicationService class of the ABP Framework and implements the ITodoAppService that was defined before. ABP provides default generic repositories for the entities. We can use them to perform the fundamental database operations. This class injects IRepository<TodoItem, Guid>, which is the default repository for the TodoItem entity. We will use it to implement the use cases described before.

***Getting Todo Items:**

```
public async Task<List<TodoItemDto>> GetListAsync(){

  var items = await _todoItemRepository.GetListAsync();

  return items

    .Select(item => new TodoItemDto

    {

        Id = item.Id,

        Text = item.Text

    }).ToList();}
```

***Creating a New Todo Item:**

```
public async Task<TodoItemDto> CreateAsync(string text){
```

```
  var todoItem = await _todoItemRepository.InsertAsync(

     new TodoItem {Text = text}

  );



  return new TodoItemDto

  {

     Id = todoItem.Id,

     Text = todoItem.Text

  };}
```

***Deleting a Todo Item**

```
public async Task DeleteAsync(Guid id){

  await _todoItemRepository.DeleteAsync(id);}
```

Step-4:User Interface Layer(TodoApp.Blazor)

Open the Index.razor.cs file in the Pages folder of the TodoApp.Blazor project and replace the content with the following code block:

I. Index.razor.cs

```
using Microsoft.AspNetCore.Components;using System.Collections.Generic;using System.Threading.Tasks;

namespace TodoApp.Blazor.Pages{

  public partial class Index

  {

     [Inject]

     private ITodoAppService TodoAppService { get; set; }



     private List<TodoItemDto> TodoItems { get; set; } = new List<TodoItemDto>();
```

```
    private string NewTodoText { get; set; }



    protected override async Task OnInitializedAsync()

    {

        TodoItems = await TodoAppService.GetListAsync();

    }



    private async Task Create()

    {

        var result = await TodoAppService.CreateAsync(NewTodoText);

        TodoItems.Add(result);

        NewTodoText = null;

    }



    private async Task Delete(TodoItemDto todoItem)

    {

        await TodoAppService.DeleteAsync(todoItem.Id);

        await Notify.Info("Deleted the todo item.");

        TodoItems.Remove(todoItem);

    }

}}
```

This class uses ITodoAppService to perform operations for the todo items. It manipulates the TodoItems list after create and delete operations. This way, we don't need to refresh the whole todo list from the server.

Ii. Index.razor

```
@page "/"

@inherits TodoAppComponentBase<div class="container">
```

```
<Card>

    <CardHeader>

        <CardTitle>

            TODO LIST

        </CardTitle>

    </CardHeader>

    <CardBody>

        <!-- FORM FOR NEW TODO ITEMS -->

        <form id="NewItemForm" @onsubmit:preventDefault @onsubmit="() => Create()" class="row row-cols-lg-auto g-3
align-items-center">

            <div class="col-12">

             <div class="input-group">

               <input name="NewTodoText" type="text" @bind-value="@NewTodoText" class="form-control"
placeholder="enter text..." />

             </div>

            </div>

            <div class="col-12">

             <button type="submit" class="btn btn-primary">Submit</button>

            </div>

        </form>

        <!-- TODO ITEMS LIST -->

        <ul id="TodoList">

            @foreach (var todoItem in TodoItems)

            {

                <li data-id="@todoItem.Id">

                    <i class="far fa-trash-alt"
```

```
                @onclick="() => Delete(todoItem)"

                ></i> @todoItem.Text

            </li>

          }

        </ul>

    </CardBody>

  </Card></div>
```

Iii. Index.razor.css

```
#TodoList{

  list-style: none;

  margin: 0;

  padding: 0;}

#TodoList li {

  padding: 5px;

  margin: 5px 0px;

  border: 1px solid #cccccc;

  background-color: #f5f5f5;}

#TodoList li i{

  opacity: 0.5;}

#TodoList li i:hover{

  opacity: 1;

  color: #ff0000;

  cursor: pointer;}
```

@Web Application Development Tutorial:-

Part 1: Creating the Server Side

Step 1:- Create the Book Entity(Domain Project)

- **Acme.BookStore.Domain contains your entities, domain services** and other core domain objects.
- **Acme.BookStore.Domain.Shared contains constants, enums** or other domain related objects that can be shared with clients.

The main entity of the application is the Book. Create a **Books folder (namespace) in the Acme.BookStore.Domain project and add a Book class inside it:**

```
using System;using Volo.Abp.Domain.Entities.Auditing;

namespace Acme.BookStore.Books;

public class Book : AuditedAggregateRoot<Guid>{

  public string Name { get; set; }



  public BookType Type { get; set; }



  public DateTime PublishDate { get; set; }



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

Note:-

- ABP Framework has two fundamental base classes for entities: AggregateRoot and Entity. **Aggregate Root** is a Domain Driven Design concept which can be thought as a root entity that is directly queried and worked on (see the entities document for more).
- The Book entity inherits from the AuditedAggregateRoot which adds some base auditing properties (like CreationTime, CreatorId, LastModificationTime...) on top of the AggregateRoot class. ABP automatically manages these properties for you.
- Guid is the **primary key type** of the Book entity.

BookType Enum (.Domain.Shared)

The Book entity uses the BookType enum. Create a Books folder (namespace) in the Acme.BookStore.Domain.Shared project and add a BookType inside it:

```
public enum BookType{
```

```
    Undefined,
    Adventure,
    Biography,
    Dystopia,
    Fantastic,
    Horror,
    Science,
    ScienceFiction,
    Poetry}
```

Step 2:- Add the Book Entity to the DbContext(In .EntityFrameworkCore)

EF Core requires that you relate the entities with your DbContext. The easiest way to do so is adding a DbSet property to the BookStoreDbContext class in the Acme.BookStore.EntityFrameworkCore project, as shown below:

```
public class BookStoreDbContext : AbpDbContext<BookStoreDbContext>{
    public DbSet<Book> Books { get; set; }
    //...}
```

Navigate to the OnModelCreating method in the BookStoreDbContext class and add the mapping code for the Book entity:

```
using Acme.BookStore.Books;...
namespace Acme.BookStore.EntityFrameworkCore;
public class BookStoreDbContext :
    AbpDbContext<BookStoreDbContext>,
    IIdentityDbContext,
    ITenantManagementDbContext{
    ...

    protected override void OnModelCreating(ModelBuilder builder)
    {
```

```
base.OnModelCreating(builder);



    /* Include modules to your migration db context */



    builder.ConfigurePermissionManagement();

    ...



    /* Configure your own tables/entities inside here */



    builder.Entity<Book>(b =>

    {

        b.ToTable(BookStoreConsts.DbTablePrefix + "Books",

            BookStoreConsts.DbSchema);

        b.ConfigureByConvention(); //auto configure for the base class props

        b.Property(x => x.Name).IsRequired().HasMaxLength(128);

    });

}}
```

Step 3:- Add Database Migration

pen a command-line terminal in the directory of the Acme.BookStore.EntityFrameworkCore project and type the following command:

```
dotnet ef migrations add Created_Book_Entity
```

If you are using Visual Studio, you may want to use the Add-Migration Created_Book_Entity and Update-Database commands in the *Package Manager Console (PMC)*. In this case, **ensure that Acme.BookStore.EntityFrameworkCore is the startup project in Visual Studio and Acme.BookStore.EntityFrameworkCore is the *Default Project* in PMC.**

Step: 4:- Add Sample Seed Data

Create a class that implements the **IDataSeedContributor interface in the *.Domain** project by copying the following code:

```
sing System;using System.Threading.Tasks;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> _bookRepository;



  public BookStoreDataSeederContributor(IRepository<Book, Guid> bookRepository)

  {

    _bookRepository = bookRepository;

  }



  public async Task SeedAsync(DataSeedContext context)

  {

    if (await _bookRepository.GetCountAsync() <= 0)

    {

      await _bookRepository.InsertAsync(

        new Book

        {

          Name = "1984",

          Type = BookType.Dystopia,

          PublishDate = new DateTime(1949, 6, 8),

          Price = 19.84f

        },

        autoSave: true

      );
```

```
        await _bookRepository.InsertAsync(
            new Book
            {
                Name = "The Hitchhiker's Guide to the Galaxy",
                Type = BookType.ScienceFiction,
                PublishDate = new DateTime(1995, 9, 27),
                Price = 42.0f
            },
            autoSave: true
        );
    }
}}
```

Step-5: Update the Database

Run the Acme.BookStore.DbMigrator application to update the database:

Step-6: Create the Application Service

- **Acme.BookStore.Application.Contracts contains your DTOs and application service interfaces.**
- **Acme.BookStore.Application contains the implementations of your application services**.

I. BookDto:

CrudAppService base class requires to define the fundamental DTOs for the entity. Create a **Books folder (namespace) in the Acme.BookStore.Application.Contracts** project and add a BookDto class inside it:

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

namespace Acme.BookStore.Books;

public class BookDto : AuditedEntityDto<Guid>{

    public string Name { get; set; }



    public BookType Type { get; set; }
```

```
  public DateTime PublishDate { get; set; }



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

Note:

- The BookDto is derived from the AuditedEntityDto<Guid> which has audit properties just like the Book entity defined above.

Create AutoMapper (.Application)

It will be needed to map the Book entities to the BookDto objects while returning books to the presentation layer. AutoMapper library can automate this conversion when you define the proper mapping. The startup template comes with AutoMapper pre-configured.

So, you can just define the mapping in the BookStoreApplicationAutoMapperProfile class in the Acme.BookStore.Application project:

```
using Acme.BookStore.Books;using AutoMapper;

namespace Acme.BookStore;

public class BookStoreApplicationAutoMapperProfile : Profile{

  public BookStoreApplicationAutoMapperProfile()

  {

    CreateMap<Book, BookDto>();

  }
```

Ii. CreateUpdateBookDto:-

Create a CreateUpdateBookDto class in the Books folder (namespace) of the Acme.BookStore.Application.Contracts project:

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

namespace Acme.BookStore.Books;

public class CreateUpdateBookDto{

  [Required]

  [StringLength(128)]
```

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



    [Required]

    public BookType Type { get; set; } = BookType.Undefined;



    [Required]

    [DataType(DataType.Date)]

    public DateTime PublishDate { get; set; } = DateTime.Now;



    [Required]

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

This DTO class is used to get a book information from the user interface while creating or updating the book.

As **done to the BookDto above, we should define the mapping from the CreateUpdateBookDto object to the Book entity. The final class will be as shown below:**

```
using Acme.BookStore.Books;using AutoMapper;

namespace Acme.BookStore;

                                                                public class BookStoreApplicationAutoMapperProfile : Profile{

    public BookStoreApplicationAutoMapperProfile()

    {

        CreateMap<Book, BookDto>();

        CreateMap<CreateUpdateBookDto, Book>();

    }}
```

Iii. IBookAppService(Application.Contracts):

Next step is to define an interface for the application service. **Create an IBookAppService interface in the Books folder (namespace) of the Acme.BookStore.Application.Contracts project**

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

```
namespace Acme.BookStore.Books;

public interface IBookAppService :

  ICrudAppService< //Defines CRUD methods

      BookDto, //Used to show books

      Guid, //Primary key of the book entity

      PagedAndSortedResultRequestDto, //Used for paging/sorting

      CreateUpdateBookDto> //Used to create/update a book{

}
```

Note:-

- Defining interfaces for the application services **are not required** by the framework. However, it's suggested as a best practice.
- **ICrudAppService defines common CRUD methods: GetAsync, GetListAsync, CreateAsync, UpdateAsync and DeleteAsync. It's not required to extend it. Instead, you could inherit from the empty IApplicationService interface and define your own methods manually (which will be done for the authors in the next parts).**

There are some variations of the ICrudAppService where you can use separated DTOs for each method (like using different DTOs for create and update).

Iv. BookAppService(.Application)

It is time to implement the IBookAppService interface. Create a new class, named BookAppService in the Books namespace (folder) of the Acme.BookStore.Application project

```
using System;using Volo.Abp.Application.Dtos;using Volo.Abp.Application.Services;using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books;

public class BookAppService :

  CrudAppService<

      Book, //The Book entity

      BookDto, //Used to show books

      Guid, //Primary key of the book entity

      PagedAndSortedResultRequestDto, //Used for paging/sorting

      CreateUpdateBookDto>, //Used to create/update a book
```

```
IBookAppService //implement the IBookAppService{

public BookAppService(IRepository<Book, Guid> repository)

    : base(repository)

{



}}
```

- BookAppService is derived from CrudAppService<...> which implements all the CRUD (create, read, update, delete) methods defined by the ICrudAppService.
-
- BookAppService injects IRepository<Book, Guid> which is the default repository for the Book entity. ABP automatically creates default repositories for each aggregate root (or entity). See the repository document.
-
- BookAppService uses IObjectMapper service (see) to map the Book objects to the BookDto objects and CreateUpdateBookDto objects to the Book objects. The Startup template uses the AutoMapper library as the object mapping provider. We have defined the mappings before, so it will work as expected.

If you try to execute the [GET] /api/app/book API to get a list of books, the server returns such a JSON result:


```
{

"totalCount": 2,

"items": [

 {

  "name": "The Hitchhiker's Guide to the Galaxy",

  "type": 7,

  "publishDate": "1995-09-27T00:00:00",

  "price": 42,

  "lastModificationTime": null,

  "lastModifierId": null,

  "creationTime": "2020-07-03T21:04:18.4607218",

  "creatorId": null,

  "id": "86100bb6-cbc1-25be-6643-39f62806969c"

 },
```

```
  {
    "name": "1984",
    "type": 3,
    "publishDate": "1949-06-08T00:00:00",
    "price": 19.84,
    "lastModificationTime": null,
    "lastModifierId": null,
    "creationTime": "2020-07-03T21:04:18.3174016",
    "creatorId": null,
    "id": "41055277-cce8-37d7-bb37-39f62806960b"
  }
 ]}
```

Part 2: The Book List Page

1. Localization(.Domain.Shared)

Localization texts are located under the Localization/BookStore folder of the Acme.BookStore.Domain.Shared project:

Open the en.json (the English translations) file and change the content as shown below:

```
{
 "Culture": "en",
 "Texts": {
   "Menu:Home": "Home",
   "Welcome": "Welcome",
```

```
  "LongWelcomeMessage": "Welcome to the application. This is a startup project based on the ABP framework. For more information, visit abp.io.",

  "Menu:BookStore": "Book Store",

  "Menu:Books": "Books",

  "Actions": "Actions",

  "Close": "Close",

  "Delete": "Delete",

  "Edit": "Edit",

  "PublishDate": "Publish date",

  "NewBook": "New book",

  "Name": "Name",

  "Type": "Type",

  "Price": "Price",

  "CreationTime": "Creation time",

  "AreYouSure": "Are you sure?",

  "AreYouSureToDelete": "Are you sure you want to delete this item?",

  "Enum:BookType.0": "Undefined",

  "Enum:BookType.1": "Adventure",

  "Enum:BookType.2": "Biography",

  "Enum:BookType.3": "Dystopia",

  "Enum:BookType.4": "Fantastic",

  "Enum:BookType.5": "Horror",

  "Enum:BookType.6": "Science",

  "Enum:BookType.7": "Science fiction",

  "Enum:BookType.8": "Poetry"

}}
```

2. Create a Books Page(.Blazor)

It's time to create something visible and usable! Right click on the Pages folder under the Acme.BookStore.Blazor project and add a new razor component, named Books.razo

```
@page "/books"

<h2>Books</h2>



@code {



}
```

3. Add the Books Page to the Main Menu

Open the BookStoreMenuContributor class in the Blazor project add the following code to the end of the ConfigureMainMenuAsync method.

```
context.Menu.AddItem(

  new ApplicationMenuItem(

    "BooksStore",

    l["Menu:BookStore"],

    icon: "fa fa-book"

  ).AddItem(

    new ApplicationMenuItem(

      "BooksStore.Books",

      l["Menu:Books"],

      url: "/books"

    )

  ));
```

Run the project, login to the application with the username admin and the password 1q2w3E* and see that the new menu item has been added to the main menu:

3. Book List:

ABP Framework provides a generic base class - AbpCrudPageBase<...>, to create CRUD style pages. This base class is compatible with the ICrudAppService that was used to build the IBookAppService. So, we can inherit from the AbpCrudPageBase to automate the code behind for the standard CRUD stuff.

Open the Books.razor and replace the content as the following:

```
@page "/books"

@using Volo.Abp.Application.Dtos

@using Acme.BookStore.Books

@using Acme.BookStore.Localization

@using Microsoft.Extensions.Localization

@inject IStringLocalizer<BookStoreResource> L

@inherits AbpCrudPageBase<IBookAppService, BookDto, Guid, PagedAndSortedResultRequestDto, CreateUpdateBookDto>

<Card>

  <CardHeader>

    <h2>@L["Books"]</h2>

  </CardHeader>

  <CardBody>

    <DataGrid TItem="BookDto"

              Data="Entities"

              ReadData="OnDataGridReadAsync"

              TotalItems="TotalCount"

              ShowPager="true"

              PageSize="PageSize">

      <DataGridColumns>

        <DataGridColumn TItem="BookDto"
```

```
                Field="@nameof(BookDto.Name)"
                Caption="@L["Name"]"></DataGridColumn>
        <DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.Type)"
                Caption="@L["Type"]">
            <DisplayTemplate>
                @L[$"Enum:BookType.{context.Type}"]
            </DisplayTemplate>
        </DataGridColumn>
        <DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.PublishDate)"
                Caption="@L["PublishDate"]">
            <DisplayTemplate>
                @context.PublishDate.ToShortDateString()
            </DisplayTemplate>
        </DataGridColumn>
        <DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.Price)"
                Caption="@L["Price"]">
        </DataGridColumn>
        <DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.CreationTime)"
                Caption="@L["CreationTime"]">
            <DisplayTemplate>
                @context.CreationTime.ToLongDateString()
```

```
            </DisplayTemplate>

        </DataGridColumn>

    </DataGridColumns>

</DataGrid>

</CardBody></Card>
```

- **Inherited from AbpCrudPageBase<IBookAppService, BookDto, Guid, PagedAndSortedResultRequestDto, CreateUpdateBookDto> which implements all the CRUD details for us.**
- Entities, TotalCount, PageSize, OnDataGridReadAsync are defined in the base class.
- **Injected IStringLocalizer<BookStoreResource> (as L object) and used for localization.**

@Run the Final Application

Part 3: Creating, Updating and Deleting Books

1. Creating a New Book:

In this section, you will learn how to create a new modal dialog form to create a new book. Since we've inherited from the **AbpCrudPageBase, we only need to develop the view part.**

I. Add a "New Button" Button:

Open the Books.razor and replace the <CardHeader> section with the following code:

```
<CardHeader>

  <Row Class="justify-content-between">

    <Column ColumnSize="ColumnSize.IsAuto">

      <h2>@L["Books"]</h2>

    </Column>

    <Column ColumnSize="ColumnSize.IsAuto">

      <Button Color="Color.Primary"

              Clicked="OpenCreateModalAsync">@L["NewBook"]</Button>

    </Column>

  </Row></CardHeader>
```

Ii. Book Creation Modal:

Open the Books.razor and add the following code to the end of the page:

```
<Modal @ref="@CreateModal">
  <ModalBackdrop />
  <ModalContent IsCentered="true">
    <Form>
      <ModalHeader>
        <ModalTitle>@L["NewBook"]</ModalTitle>
        <CloseButton Clicked="CloseCreateModalAsync"/>
      </ModalHeader>
      <ModalBody>
        <Validations @ref="@CreateValidationsRef" Model="@NewEntity" ValidateOnLoad="false">
          <Validation MessageLocalizer="@LH.Localize">
            <Field>
              <FieldLabel>@L["Name"]</FieldLabel>
              <TextEdit @bind-Text="@NewEntity.Name">
                <Feedback>
                  <ValidationError/>
                </Feedback>
              </TextEdit>
            </Field>
          </Validation>
          <Field>
            <FieldLabel>@L["Type"]</FieldLabel>
```

```
            <Select TValue="BookType" @bind-SelectedValue="@NewEntity.Type">
                @foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))
                {
                    <SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">
                        @L[$"Enum:BookType.{bookTypeValue}"]
                    </SelectItem>
                }
            </Select>
        </Field>
        <Field>
            <FieldLabel>@L["PublishDate"]</FieldLabel>
            <DateEdit TValue="DateTime" @bind-Date="NewEntity.PublishDate"/>
        </Field>
        <Field>
            <FieldLabel>@L["Price"]</FieldLabel>
            <NumericEdit TValue="float" @bind-Value="NewEntity.Price"/>
        </Field>
    </Validations>
</ModalBody>
<ModalFooter>
    <Button Color="Color.Secondary"
            Clicked="CloseCreateModalAsync">@L["Cancel"]</Button>
    <Button Color="Color.Primary"
            Type="@ButtonType.Submit"
            PreventDefaultOnSubmit="true"
```

```
            Clicked="CreateEntityAsync">@L["Save"]</Button>

        </ModalFooter>

    </Form>

  </ModalContent></Modal>
```

Iii. Updating a Book:

Actions Dropdown:- Open the Books.razor and add the following DataGridEntityActionsColumn section inside the DataGridColumns as the first item:

```
<DataGridEntityActionsColumn TItem="BookDto" @ref="@EntityActionsColumn">

  <DisplayTemplate>

    <EntityActions TItem="BookDto" EntityActionsColumn="@EntityActionsColumn">

      <EntityAction TItem="BookDto"

                Text="@L["Edit"]"

                Clicked="() => OpenEditModalAsync(context)" />

    </EntityActions>

  </DisplayTemplate></DataGridEntityActionsColumn>
```

- OpenEditModalAsync is defined in the base class which takes the entity (book) to edit.

The DataGridEntityActionsColumn component is used to show an "Actions" dropdown for each row in the DataGrid.
The DataGridEntityActionsColumn shows a **single button** instead of a dropdown if there is only one available action inside it:

Iv. Edit Modal:

We can now define a modal to edit the book. Add the following code to the end of the Books.razor page:

```
<Modal @ref="@EditModal">

  <ModalBackdrop />

  <ModalContent IsCentered="true">

    <Form>

      <ModalHeader>

        <ModalTitle>@EditingEntity.Name</ModalTitle>
```

```
        <CloseButton Clicked="CloseEditModalAsync"/>
    </ModalHeader>
    <ModalBody>
        <Validations @ref="@EditValidationsRef" Model="@NewEntity" ValidateOnLoad="false">
            <Validation MessageLocalizer="@LH.Localize">
                <Field>
                    <FieldLabel>@L["Name"]</FieldLabel>
                    <TextEdit @bind-Text="@EditingEntity.Name">
                        <Feedback>
                            <ValidationError/>
                        </Feedback>
                    </TextEdit>
                </Field>
            </Validation>
            <Field>
                <FieldLabel>@L["Type"]</FieldLabel>
                <Select TValue="BookType" @bind-SelectedValue="@EditingEntity.Type">
                    @foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))
                    {
                        <SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">
                            @L[$"Enum:BookType.{bookTypeValue}"]
                        </SelectItem>
                    }
                </Select>
            </Field>
```

```
            <Field>
                <FieldLabel>@L["PublishDate"]</FieldLabel>
                <DateEdit TValue="DateTime" @bind-Date="EditingEntity.PublishDate"/>
            </Field>
            <Field>
                <FieldLabel>@L["Price"]</FieldLabel>
                <NumericEdit TValue="float" @bind-Value="EditingEntity.Price"/>
            </Field>
        </Validations>
    </ModalBody>
    <ModalFooter>
        <Button Color="Color.Secondary"
                Clicked="CloseEditModalAsync">@L["Cancel"]</Button>
        <Button Color="Color.Primary"
                Type="@ButtonType.Submit"
                PreventDefaultOnSubmit="true"
                Clicked="UpdateEntityAsync">@L["Save"]</Button>
    </ModalFooter>
  </Form>
</ModalContent></Modal>
```

#AutoMapper Configuration:-

The base AbpCrudPageBase uses the object to object mapping system to convert an incoming BookDto object to a CreateUpdateBookDto object. So, we need to define the mapping.

Open the BookStoreBlazorAutoMapperProfile inside the Acme.BookStore.Blazor project and change the content as the following:

```
using Acme.BookStore.Books;using AutoMapper;

namespace Acme.BookStore.Blazor;

public class BookStoreBlazorAutoMapperProfile : Profile{

  public BookStoreBlazorAutoMapperProfile()

  {

    CreateMap<BookDto, CreateUpdateBookDto>();

  }}
```

V. Deleting a Book

Open the Books.razor page and add the following EntityAction code under the "Edit" action inside EntityActions

```
<EntityAction TItem="BookDto"

        Text="@L["Delete"]"

        Clicked="() => DeleteEntityAsync(context)"

        ConfirmationMessage="() => GetDeleteConfirmationMessage(context)" />
```

- DeleteEntityAsync is defined in the base class that deletes the entity by performing a call to the server.
- ConfirmationMessage is a callback to show a confirmation message before executing the action.
- GetDeleteConfirmationMessage is defined in the base class. You can override this method (or pass another value to the ConfirmationMessage parameter) to customize the localization message.

*** Full CRUD UI Code:-

```
@page "/books"

@using Volo.Abp.Application.Dtos

@using Acme.BookStore.Books

@using Acme.BookStore.Localization

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Components.Web

@inject IStringLocalizer<BookStoreResource> L

@inject AbpBlazorMessageLocalizerHelper<BookStoreResource> LH
```

```
@inherits AbpCrudPageBase<IBookAppService, BookDto, Guid, PagedAndSortedResultRequestDto, CreateUpdateBookDto>

<Card>
  <CardHeader>
    <Row Class="justify-content-between">
      <Column ColumnSize="ColumnSize.IsAuto">
        <h2>@L["Books"]</h2>
      </Column>
      <Column ColumnSize="ColumnSize.IsAuto">
        <Button Color="Color.Primary"
                Clicked="OpenCreateModalAsync">@L["NewBook"]</Button>
      </Column>
    </Row>
  </CardHeader>
  <CardBody>
    <DataGrid TItem="BookDto"
              Data="Entities"
              ReadData="OnDataGridReadAsync"
              CurrentPage="CurrentPage"
              TotalItems="TotalCount"
              ShowPager="true"
              PageSize="PageSize">
      <DataGridColumns>
        <DataGridEntityActionsColumn TItem="BookDto" @ref="@EntityActionsColumn">
          <DisplayTemplate>
            <EntityActions TItem="BookDto" EntityActionsColumn="@EntityActionsColumn">
```

```
            <EntityAction TItem="BookDto"
                          Text="@L["Edit"]"
                          Clicked="() => OpenEditModalAsync(context)" />
            <EntityAction TItem="BookDto"
                          Text="@L["Delete"]"
                          Clicked="() => DeleteEntityAsync(context)"
                          ConfirmationMessage="()=>GetDeleteConfirmationMessage(context)" />
        </EntityActions>
    </DisplayTemplate>
</DataGridEntityActionsColumn>
<DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.Name)"
                Caption="@L["Name"]"></DataGridColumn>
<DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.Type)"
                Caption="@L["Type"]">
    <DisplayTemplate>
        @L[$"Enum:BookType.{context.Type}"]
    </DisplayTemplate>
</DataGridColumn>
<DataGridColumn TItem="BookDto"
                Field="@nameof(BookDto.PublishDate)"
                Caption="@L["PublishDate"]">
    <DisplayTemplate>
        @context.PublishDate.ToShortDateString()
```

```
            </DisplayTemplate>
        </DataGridColumn>
        <DataGridColumn TItem="BookDto"
                        Field="@nameof(BookDto.Price)"
                        Caption="@L["Price"]">
        </DataGridColumn>
        <DataGridColumn TItem="BookDto"
                        Field="@nameof(BookDto.CreationTime)"
                        Caption="@L["CreationTime"]">
            <DisplayTemplate>
                @context.CreationTime.ToLongDateString()
            </DisplayTemplate>
        </DataGridColumn>
    </DataGridColumns>
  </DataGrid>
 </CardBody></Card>
<Modal @ref="@CreateModal">
 <ModalBackdrop />
 <ModalContent IsCentered="true">
  <Form>
    <ModalHeader>
      <ModalTitle>@L["NewBook"]</ModalTitle>
      <CloseButton Clicked="CloseCreateModalAsync"/>
    </ModalHeader>
    <ModalBody>
```

```
<Validations @ref="@CreateValidationsRef" Model="@NewEntity" ValidateOnLoad="false">
  <Validation MessageLocalizer="@LH.Localize">
    <Field>
      <FieldLabel>@L["Name"]</FieldLabel>
      <TextEdit @bind-Text="@NewEntity.Name">
        <Feedback>
          <ValidationError/>
        </Feedback>
      </TextEdit>
    </Field>
  </Validation>
  <Field>
    <FieldLabel>@L["Type"]</FieldLabel>
    <Select TValue="BookType" @bind-SelectedValue="@NewEntity.Type">
      @foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))
      {
        <SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">
          @L[$"Enum:BookType.{bookTypeValue}"]
        </SelectItem>
      }
    </Select>
  </Field>
  <Field>
    <FieldLabel>@L["PublishDate"]</FieldLabel>
    <DateEdit TValue="DateTime" @bind-Date="NewEntity.PublishDate"/>
```

```
            </Field>

            <Field>

                <FieldLabel>@L["Price"]</FieldLabel>

                <NumericEdit TValue="float" @bind-Value="NewEntity.Price"/>

            </Field>

        </Validations>

    </ModalBody>

    <ModalFooter>

        <Button Color="Color.Secondary"

                Clicked="CloseCreateModalAsync">@L["Cancel"]</Button>

        <Button Color="Color.Primary"

                Type="@ButtonType.Submit"

                PreventDefaultOnSubmit="true"

                Clicked="CreateEntityAsync">@L["Save"]</Button>

    </ModalFooter>

  </Form>

 </ModalContent></Modal>

<Modal @ref="@EditModal">

 <ModalBackdrop />

 <ModalContent IsCentered="true">

  <Form>

    <ModalHeader>

        <ModalTitle>@EditingEntity.Name</ModalTitle>

        <CloseButton Clicked="CloseEditModalAsync"/>

    </ModalHeader>
```

```
    <ModalBody>

        <Validations @ref="@EditValidationsRef" Model="@NewEntity" ValidateOnLoad="false">

            <Validation MessageLocalizer="@LH.Localize">

                <Field>

                    <FieldLabel>@L["Name"]</FieldLabel>

                    <TextEdit @bind-Text="@EditingEntity.Name">

                        <Feedback>

                            <ValidationError/>

                        </Feedback>

                    </TextEdit>

                </Field>

            </Validation>

            <Field>

                <FieldLabel>@L["Type"]</FieldLabel>

                <Select TValue="BookType" @bind-SelectedValue="@EditingEntity.Type">

                    @foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))

                    {

                        <SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">

                            @L[$"Enum:BookType.{bookTypeValue}"]

                        </SelectItem>

                    }

                </Select>

            </Field>

            <Field>

                <FieldLabel>@L["PublishDate"]</FieldLabel>
```

```
            <DateEdit TValue="DateTime" @bind-Date="EditingEntity.PublishDate"/>

          </Field>

          <Field>

            <FieldLabel>@L["Price"]</FieldLabel>

            <NumericEdit TValue="float" @bind-Value="EditingEntity.Price"/>

          </Field>

        </Validations>

      </ModalBody>

      <ModalFooter>

        <Button Color="Color.Secondary"

            Clicked="CloseEditModalAsync">@L["Cancel"]</Button>

        <Button Color="Color.Primary"

            Type="@ButtonType.Submit"

            PreventDefaultOnSubmit="true"

            Clicked="UpdateEntityAsync">@L["Save"]</Button>

      </ModalFooter>

    </Form>

</ModalContent></Modal>
```

Part 5: Authorization(Acme.BookStore.Application.Contracts)

1. Permissions:

ABP Framework provides an authorization system based on the ASP.NET Core's authorization infrastructure. One major feature added on top of the standard authorization infrastructure is the **permission system** which allows to define permissions and enable/disable per role, user or client.

I. Permission Names:

A permission must have a unique name (a string). The best way is to define it as a const, so we can reuse the permission name.

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 + ".Books";

    public const string Create = Default + ".Create";

    public const string Edit = Default + ".Edit";

    public const string Delete = Default + ".Delete";

  }}
```

This is a hierarchical way of defining permission names. For example, "create book" permission name was defined as BookStore.Books.Create. ABP doesn't force you to a structure, but we find this way useful.

Ii. Permission Definitions:

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

```
using Acme.BookStore.Localization;using Volo.Abp.Authorization.Permissions;using Volo.Abp.Localization;
```

```
namespace Acme.BookStore.Permissions;

public class BookStorePermissionDefinitionProvider : PermissionDefinitionProvider{

  public override void Define(IPermissionDefinitionContext context)

  {

    var bookStoreGroup = context.AddGroup(BookStorePermissions.GroupName, L("Permission:BookStore"));



    var booksPermission = bookStoreGroup.AddPermission(BookStorePermissions.Books.Default, L("Permission:Books"));

    booksPermission.AddChild(BookStorePermissions.Books.Create, L("Permission:Books.Create"));

    booksPermission.AddChild(BookStorePermissions.Books.Edit, L("Permission:Books.Edit"));

    booksPermission.AddChild(BookStorePermissions.Books.Delete, L("Permission:Books.Delete"));

  }



  private static LocalizableString L(string name)

  {

    return LocalizableString.Create<BookStoreResource>(name);

  }}
```

This class defines a **permission group** (to group permissions on the UI, will be seen below) and **4 permissions** inside this group. Also, **Create**, **Edit** and **Delete** are children of the BookStorePermissions.Books.Default permission. A child permission can be selected **only if the parent was selected**.

Finally, edit the localization file (en.json under the Localization/BookStore folder of the Acme.BookStore.Domain.Shared project) to define the localization keys used above:

```
"Permission:BookStore": "Book Store","Permission:Books": "Book Management","Permission:Books.Create": "Creating new books","Permission:Books.Edit": "Editing the books","Permission:Books.Delete": "Deleting the books"
```

Iii. Permission Management UI

Grant the permissions you want and save the modal.

> **Tip**: New permissions are automatically granted to the admin role if you run the Acme.BookStore.DbMigrator application.

2. Authorization:

I. Application Layer & HTTP API:

Open the BookAppService class and set the policy names as the permission names defined above:

Explain

```
using System;using Acme.BookStore.Permissions;using Volo.Abp.Application.Dtos;using Volo.Abp.Application.Services;using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books;

public class BookAppService :

  CrudAppService<

    Book, //The Book entity

    BookDto, //Used to show books

    Guid, //Primary key of the book entity

    PagedAndSortedResultRequestDto, //Used for paging/sorting

    CreateUpdateBookDto>, //Used to create/update a book

  IBookAppService //implement the IBookAppService{

  public BookAppService(IRepository<Book, Guid> repository)

    : base(repository)

  {

    GetPolicyName = BookStorePermissions.Books.Default;

    GetListPolicyName = BookStorePermissions.Books.Default;

    CreatePolicyName = BookStorePermissions.Books.Create;

    UpdatePolicyName = BookStorePermissions.Books.Edit;

    DeletePolicyName = BookStorePermissions.Books.Delete;

  }}
```

Added code to the constructor. Base CrudAppService automatically uses these permissions on the CRUD operations. This makes the **application service** secure, but also makes the **HTTP API** secure since this service is automatically used as an HTTP API as explained before (see auto API controllers).

Part 6: Authors: Domain Layer

Introduction:

- Used the CrudAppService base class instead of manually developing an application service for standard create, read, update and delete operations.
- Used generic repositories to completely automate the database layer

I. The Author Entity:

Create an Authors folder (namespace) in the Acme.BookStore.Domain project and add an Author class inside it:

```
using System;using JetBrains.Annotations;using Volo.Abp;using Volo.Abp.Domain.Entities.Auditing;

namespace Acme.BookStore.Authors;

public class Author : FullAuditedAggregateRoot<Guid>{

  public string Name { get; private set; }

  public DateTime BirthDate { get; set; }

  public string ShortBio { get; set; }



  private Author()

  {

    /* This constructor is for deserialization / ORM purpose */

  }



  internal Author(

    Guid id,

    [NotNull] string name,

    DateTime birthDate,
```

```
    [CanBeNull] string shortBio = null)
    : base(id)
{
    SetName(name);
    BirthDate = birthDate;
    ShortBio = shortBio;
}

internal Author ChangeName([NotNull] string name)
{
    SetName(name);
    return this;
}

private void SetName([NotNull] string name)
{
    Name = Check.NotNullOrWhiteSpace(
        name,
        nameof(name),
        maxLength: AuthorConsts.MaxNameLength
    );
}}
```

AuthorConsts is a simple class that is located under the Authors namespace (folder) of the Acme.BookStore.Domain.Shared project:

```
namespace Acme.BookStore.Authors;

public static class AuthorConsts{

   public const int MaxNameLength = 64;}
```

Ii. AuthorManager: The Domain Service:

Author constructor and ChangeName methods are internal, so they can be used only in the domain layer. Create an AuthorManager class in the Authors folder (namespace) of the Acme.BookStore.Domain project:

```
using System;using System.Threading.Tasks;using JetBrains.Annotations;using Volo.Abp;using Volo.Abp.Domain.Services;

namespace Acme.BookStore.Authors;

public class AuthorManager : DomainService{

   private readonly IAuthorRepository _authorRepository;



   public AuthorManager(IAuthorRepository authorRepository)

   {

      _authorRepository = authorRepository;

   }



   public async Task<Author> CreateAsync(

      [NotNull] string name,

      DateTime birthDate,

      [CanBeNull] string shortBio = null)

   {

      Check.NotNullOrWhiteSpace(name, nameof(name));



      var existingAuthor = await _authorRepository.FindByNameAsync(name);
```

```
    if (existingAuthor != null)
    {
        throw new AuthorAlreadyExistsException(name);
    }

    return new Author(
        GuidGenerator.Create(),
        name,
        birthDate,
        shortBio
    );
}

public async Task ChangeNameAsync(
    [NotNull] Author author,
    [NotNull] string newName)
{
    Check.NotNull(author, nameof(author));
    Check.NotNullOrWhiteSpace(newName, nameof(newName));

    var existingAuthor = await _authorRepository.FindByNameAsync(newName);
    if (existingAuthor != null && existingAuthor.Id != author.Id)
    {
        throw new AuthorAlreadyExistsException(newName);
    }
```

```
    author.ChangeName(newName);

  }}
```

- AuthorManager forces to create an author and change name of an author in a controlled way. The application layer (will be introduced later) will use these methods.

DDD tip: Do not introduce domain service methods unless they are really needed and perform some core business rules. For this case, we needed this service to be able to force the unique name constraint

Both methods checks if there is already an author with the given name and throws a special business exception, AuthorAlreadyExistsException, defined in the Acme.BookStore.Domain project (in the Authors folder) as shown below:

```
using Volo.Abp;

namespace Acme.BookStore.Authors;

public class AuthorAlreadyExistsException : BusinessException{

  public AuthorAlreadyExistsException(string name)

    : base(BookStoreDomainErrorCodes.AuthorAlreadyExists)

  {

    WithData("name", name);

  }}
```

BusinessException is a special exception type. It is a good practice to throw domain related exceptions when needed. It is automatically handled by the ABP Framework and can be easily localized. WithData(...) method is used to provide additional data to the exception object that will later be used on the localization message or for some other purpose.

Open the BookStoreDomainErrorCodes in the Acme.BookStore.Domain.Shared project and change as shown below:

```
namespace Acme.BookStore;

public static class BookStoreDomainErrorCodes{

  public const string AuthorAlreadyExists = "BookStore:00001";}
```

This is a unique string represents the error code thrown by your application and can be handled by client applications. For users, you probably want to localize it. Open the Localization/BookStore/en.json inside the Acme.BookStore.Domain.Shared project and add the following entry:

```
"BookStore:00001": "There is already an author with the same name: {name}"
```

Whenever you throw an AuthorAlreadyExistsException, the end user will see a nice error message on the UI.

Iii. IAuthorRepository:

AuthorManager injects the IAuthorRepository, so we need to define it. Create this new interface in the Authors folder (namespace) of the Acme.BookStore.Domain project:

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

namespace Acme.BookStore.Authors;

public interface IAuthorRepository : IRepository<Author, Guid>{

  Task<Author> FindByNameAsync(string name);



  Task<List<Author>> GetListAsync(

    int skipCount,

    int maxResultCount,

    string sorting,

    string filter = null

  );}
```

- IAuthorRepository extends the standard IRepository<Author, Guid> interface, so all the standard repository methods will also be available for the IAuthorRepository.
- FindByNameAsync was used in the AuthorManager to query an author by name.
- GetListAsync will be used in the application layer to get a listed, sorted and filtered list of authors to show on the UI.