The software architecture

based on [Domain Driven Design] principles and patterns

Contents

[1 Quick Start 9](#_Toc143680733)

[1.1 With single-layer solution 9](#_Toc143680734)

[1.2 With layered architecture 27](#_Toc143680735)

[2 Getting Started 45](#_Toc143680736)

[2.1 Web Application - Layered Architecture 45](#_Toc143680737)

[2.1.1 1: Setup Your Development Environment 45](#_Toc143680738)

[2.1.2 2: Creating a New Solution 46](#_Toc143680739)

[2.1.3 3: Running the Solution 48](#_Toc143680740)

[2.2 Web Application - Single-Layered Architecture 53](#_Toc143680741)

[2.2.1 1: Setup Your Development Environment 54](#_Toc143680742)

[2.2.2 2: Creating a New Solution 55](#_Toc143680743)

[2.2.3 3: Running the Solution 56](#_Toc143680744)

[2.3 Console Application 58](#_Toc143680745)

[2.4 WPF Application 59](#_Toc143680746)

[2.5 MAUI 59](#_Toc143680747)

[2.6 Empty Web Project 60](#_Toc143680748)

[3 Tutorials 63](#_Toc143680749)

[3.1 Web Application Development 63](#_Toc143680750)

[3.1.1 1: Creating the Server Side 63](#_Toc143680751)

[3.1.2 2: The Book List Page 73](#_Toc143680752)

[3.1.3 3: Creating, Updating and Deleting Books 88](#_Toc143680753)

[3.1.4 4: Integration Tests 118](#_Toc143680754)

[3.1.5 5: Authorization 123](#_Toc143680755)

[3.1.6 6: Authors: Domain layer 135](#_Toc143680756)

[3.1.7 7: Authors: Database Integration 141](#_Toc143680757)

[3.1.8 8: Authors: Application Layer 146](#_Toc143680758)

[3.1.9 9: Authors: User Interface 157](#_Toc143680759)

[3.1.10 10: Book to Author Relation 181](#_Toc143680760)

[3.2 Community Articles 205](#_Toc143680761)

[3.3 Migrating from the ASP.NET Boilerplate 205](#_Toc143680762)

[4 CLI 224](#_Toc143680763)

[4.1 Examples for the new command 238](#_Toc143680764)

[5 Startup Templates 244](#_Toc143680765)

[5.1 Overall 244](#_Toc143680766)

[5.2 Application 245](#_Toc143680767)

[5.3 Application (Single Layer) 256](#_Toc143680768)

[5.4 Module 259](#_Toc143680769)

[5.5 Console 265](#_Toc143680770)

[5.6 WPF 266](#_Toc143680771)

[6 Fundamentals 267](#_Toc143680772)

[6.1 Application Startup 267](#_Toc143680773)

[6.2 Authorization 276](#_Toc143680774)

[6.3 Caching 287](#_Toc143680775)

[6.3.1 Entity Cache 294](#_Toc143680776)

[6.3.2 Redis Cache 297](#_Toc143680777)

[6.4 Configuration 298](#_Toc143680778)

[6.4.1 Connection Strings 298](#_Toc143680779)

[6.5 Dependency Injection 302](#_Toc143680780)

[6.5.1 AutoFac Integration 313](#_Toc143680781)

[6.6 Exception Handling 316](#_Toc143680782)

[6.7 Localization 323](#_Toc143680783)

[6.8 Logging 328](#_Toc143680784)

[6.9 Object Extensions 329](#_Toc143680785)

[6.10 Options 337](#_Toc143680786)

[6.11 Settings 340](#_Toc143680787)

[6.12 Validation 345](#_Toc143680788)

[6.12.1 FluentValidation Integration 349](#_Toc143680789)

[7 Infrastructure 351](#_Toc143680790)

[7.1 Audit Logging 351](#_Toc143680791)

[7.2 Background Jobs 360](#_Toc143680792)

[7.2.1 Hangfire Integration 366](#_Toc143680793)

[7.2.2 RabbitMQ Integration 370](#_Toc143680794)

[7.2.3 Quartz Integration 373](#_Toc143680795)

[7.3 Background Workers 377](#_Toc143680796)

[7.3.1 Quartz Integration 380](#_Toc143680797)

[7.3.2 Hangfire Integration 384](#_Toc143680798)

[7.4 BLOB Storing 387](#_Toc143680799)

[7.4.1 BLOB Storing System 387](#_Toc143680800)

[7.4.2 Storage Providers 395](#_Toc143680801)

[7.5 Cancellation Token Provider 411](#_Toc143680802)

[7.6 CSRF/XSRF & Anti Forgery 413](#_Toc143680803)

[7.7 Concurrency Check 417](#_Toc143680804)

[7.8 Current User 420](#_Toc143680805)

[7.9 Data Filtering 424](#_Toc143680806)

[7.10 Data Seeding 430](#_Toc143680807)

[7.11 Distributed Locking 435](#_Toc143680808)

[7.12 Email Sending 435](#_Toc143680809)

[7.12.1 Email Sending System 435](#_Toc143680810)

[7.12.2 MailKit Integration 441](#_Toc143680811)

[7.13 Event Bus 442](#_Toc143680812)

[7.13.1 Overall 442](#_Toc143680813)

[7.13.2 Local Event Bus 442](#_Toc143680814)

[7.13.3 Distributed Event Bus 448](#_Toc143680815)

[7.14 Features 476](#_Toc143680816)

[7.15 Global Features 486](#_Toc143680817)

[7.16 GUID Generation 489](#_Toc143680818)

[7.17 Image Manipulation 492](#_Toc143680819)

[7.18 JSON 500](#_Toc143680820)

[7.19 Object to Object Mapping 502](#_Toc143680821)

[7.20 Simple State Checker 509](#_Toc143680822)

[7.21 SMS Sending 512](#_Toc143680823)

[7.22 String Encryption 514](#_Toc143680824)

[7.22.1 Text Templating 517](#_Toc143680825)

[7.23 Timing 540](#_Toc143680826)

[7.24 Virtual File System 543](#_Toc143680827)

[8 Architecture 547](#_Toc143680828)

[8.1 Modularity 547](#_Toc143680829)

[8.1.1 Basics 547](#_Toc143680830)

[8.1.2 Plug-In Modules 552](#_Toc143680831)

[8.1.3 Best Practices 558](#_Toc143680832)

[8.1.4 Customizing/Extending Modules 579](#_Toc143680833)

[8.2 Domain Driven Design 606](#_Toc143680834)

[8.2.1 Overall 606](#_Toc143680835)

[8.2.2 Domain Layer 607](#_Toc143680836)

[8.2.3 Application Layer 638](#_Toc143680837)

[8.2.4 E-Book: Implementing DDD 665](#_Toc143680838)

[8.3 Multi Tenancy 665](#_Toc143680839)

[8.4 Microservices 674](#_Toc143680840)

[9 API 675](#_Toc143680841)

[9.1 ABP Endpoints 675](#_Toc143680842)

[9.1.1 Application Configuration 675](#_Toc143680843)

[9.1.2 Application Localization 676](#_Toc143680844)

[9.2 API Versioning 677](#_Toc143680845)

[9.3 Auto API Controllers 684](#_Toc143680846)

[9.4 Dynamic C# API Clients 689](#_Toc143680847)

[9.5 Integration Services 694](#_Toc143680848)

[9.6 Static C# API Clients 697](#_Toc143680849)

[9.7 Swagger Integration 703](#_Toc143680850)

[10 User Interface 708](#_Toc143680851)

[10.1 MVC / Razor Pages 708](#_Toc143680852)

[10.1.1 Overall 708](#_Toc143680853)

[10.1.2 Navigation / Menus 708](#_Toc143680854)

[10.1.3 Forms & Validation 708](#_Toc143680855)

[10.1.4 Modals 708](#_Toc143680856)

[10.1.5 Data Tables 708](#_Toc143680857)

[10.1.6 Auto-Complete Select 708](#_Toc143680858)

[10.1.7 Page Alerts 708](#_Toc143680859)

[10.1.8 Dynamic JavaScript API Client Proxies 708](#_Toc143680860)

[10.1.9 Static JavaScript API Client Proxies 708](#_Toc143680861)

[10.1.10 Client Side Package Management 708](#_Toc143680862)

[10.1.11 Bundling & Minification 708](#_Toc143680863)

[10.1.12 Tag Helpers 708](#_Toc143680864)

[10.1.13 Widgets 708](#_Toc143680865)

[10.1.14 Toolbars 708](#_Toc143680866)

[10.1.15 Page Header 708](#_Toc143680867)

[10.1.16 Branding 708](#_Toc143680868)

[10.1.17 Layout Hooks 708](#_Toc143680869)

[10.1.18 Testing 708](#_Toc143680870)

[10.1.19 Theming 708](#_Toc143680871)

[10.1.20 JavaScript API 708](#_Toc143680872)

[10.1.21 Customize/Extend the UI 709](#_Toc143680873)

[10.1.22 Security 709](#_Toc143680874)

[10.2 Blazor 709](#_Toc143680875)

[10.2.1 Overall 709](#_Toc143680876)

[10.2.2 Navigation / Menu 709](#_Toc143680877)

[10.2.3 Localization 709](#_Toc143680878)

[10.2.4 Theming 709](#_Toc143680879)

[10.2.5 Security 709](#_Toc143680880)

[10.2.6 Services 709](#_Toc143680881)

[10.2.7 Other Components 709](#_Toc143680882)

[10.2.8 Settings 709](#_Toc143680883)

[10.2.9 Error Handling 709](#_Toc143680884)

[10.2.10 Customization / Overriding Components 709](#_Toc143680885)

[10.2.11 Global Scripts & Styles 709](#_Toc143680886)

[10.2.12 Global Features 710](#_Toc143680887)

[10.2.13 Routing 710](#_Toc143680888)

[10.2.14 PWA Configuration 710](#_Toc143680889)

[10.2.15 Layout Hooks 710](#_Toc143680890)

[10.3 Angular 710](#_Toc143680891)

[10.3.1 Quick Start 710](#_Toc143680892)

[10.3.2 Development 710](#_Toc143680893)

[10.3.3 Core Functionality 710](#_Toc143680894)

[10.3.4 Utilities 710](#_Toc143680895)

[10.3.5 Customization 711](#_Toc143680896)

[10.3.6 Components 711](#_Toc143680897)

[10.4 React Native 711](#_Toc143680898)

[10.4.1 Getting Started 711](#_Toc143680899)

[10.5 Common 711](#_Toc143680900)

[11 Data Access 711](#_Toc143680901)

[11.1 Overall 711](#_Toc143680902)

[11.2 Entity Framework Core 711](#_Toc143680903)

[11.2.1 Database Migrations 712](#_Toc143680904)

[11.2.2 Switch DBMS 712](#_Toc143680905)

[11.3 MongoDB 712](#_Toc143680906)

[11.4 Dapper 712](#_Toc143680907)

[12 Real Time 712](#_Toc143680908)

[12.1 SignalR Integration 712](#_Toc143680909)

[13 Dapr Integration 712](#_Toc143680910)

[14 Testing 712](#_Toc143680911)

[15 Deployment 712](#_Toc143680912)

[15.1 Configuring OpenIddict 712](#_Toc143680913)

[15.2 Configuring for Production 712](#_Toc143680914)

[15.3 Deploying to a Clustered Environment 712](#_Toc143680915)

[15.4 Distributed / Microservice Solutions 712](#_Toc143680916)

[15.5 Optimizing for Production 712](#_Toc143680917)

[16 Application Modules 712](#_Toc143680918)

[16.1 Overall 712](#_Toc143680919)

[16.2 Account 712](#_Toc143680920)

[16.3 Audit Logging 712](#_Toc143680921)

[16.4 Background Jobs 712](#_Toc143680922)

[16.5 CMS Kit 712](#_Toc143680923)

[16.6 Docs 712](#_Toc143680924)

[16.7 Feature Management 712](#_Toc143680925)

[16.8 Identity 712](#_Toc143680926)

[16.9 IdentityServer 712](#_Toc143680927)

[16.9.1 IdentityServer Migration Guide 712](#_Toc143680928)

[16.10 OpenIddict 713](#_Toc143680929)

[16.10.1 OpenIddict Migration Guide 713](#_Toc143680930)

[16.11 Permission Management 713](#_Toc143680931)

[16.12 Setting Management 713](#_Toc143680932)

[16.13 Tenant Management 713](#_Toc143680933)

[16.14 Virtual File Explorer 713](#_Toc143680934)

[16.15 Common 713](#_Toc143680935)

[16.15.1 Database Tables 713](#_Toc143680936)

[17 Samples 713](#_Toc143680937)

[17.1 All Samples 713](#_Toc143680938)

[17.2 eShopOnAbp 713](#_Toc143680939)

[17.3 EventHub 713](#_Toc143680940)

[17.4 Microservice Demo (legacy) 713](#_Toc143680941)

[18 Books 713](#_Toc143680942)

[18.1 Mastering ABP Framework 713](#_Toc143680943)

[18.2 Implementing Domain Driven Design 713](#_Toc143680944)

[19 Release Information 713](#_Toc143680945)

[19.1 Upgrading 713](#_Toc143680946)

[19.2 Official Packages 713](#_Toc143680947)

[19.3 Preview Releases 713](#_Toc143680948)

[19.4 Nightly Builds 713](#_Toc143680949)

[19.5 Road Map 713](#_Toc143680950)

[19.6 Migration Guides 713](#_Toc143680951)

[20 Contribution Guide 713](#_Toc143680952)

# Quick Start

# Quick Start: Overall

**\*\*Welcome to the ABP Framework\*\***. This is a single-part, quick-start tutorial to build a simple application. Start with this tutorial if you want to quickly understand how ABP Framework works.

## Select the Solution Architecture

This tutorial has multiple versions. Please select the one best fits for you:

\* **\*\*[**Single-Layer Solution**](Single-Layer/Index.md)\*\***: Creates a single-project solution. Recommended for building an application with a **\*\*simpler and easy to understand\*\*** architecture.

\* **\*\*[**Layered Solution Architecture**](Index.md)\*\***: A fully layered (multiple projects) solution based on [Domain Driven Design](../../Domain-Driven-Design.md) practices. Recommended for long-term projects that need a **\*\*maintainable and extensible\*\*** codebase.

## See Also

\* Check the [Web Application Development Tutorial](../Part-1.md) to see a real-life web application development in a layered architecture.

## With single-layer solution

# Quick Start

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

This is a single-part quick-start tutorial to build a simple todo application with the ABP Framework. Here's a screenshot from the final application:

![todo-list](../todo-list.png)

You can find the source code of the completed application [here](https://github.com/abpframework/abp-samples/tree/master/TodoApp-SingleLayer).

{{if UI=="Blazor"}}

We are currently preparing a video tutorial for Blazor UI. You can watch other tutorials for the three UI types from [here](https://www.youtube.com/playlist?list=PLsNclT2aHJcPqZxk7D4tU8LtTeCFcN\_ci).

{{else}}

This documentation has a video tutorial on **\*\*YouTube\*\***!! You can watch it here:

{{end}}

{{if UI=="MVC" && DB =="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/Z6jZSPB19iw" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

{{else if UI=="BlazorServer" && DB=="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/-ynMYXBIg4Q" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

{{else if UI=="NG" && DB=="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/Pz4YWsU7CUs" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

{{else if UI=="MVC" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/i9oDVl1J7Dk" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

{{else if UI=="BlazorServer" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/z7YGDjcsTTs" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

{{else if UI=="NG" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/LdKlIHi9S8I" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share" allowfullscreen></iframe>

{{end}}

## Pre-Requirements

\* An IDE (e.g. [Visual Studio](https://visualstudio.microsoft.com/vs/)) that supports [.NET 7.0+](https://dotnet.microsoft.com/download/dotnet) development.

\* [Node v16.x](https://nodejs.org/)

{{if DB=="Mongo"}}

\* [MongoDB Server 4.0+](https://docs.mongodb.com/manual/administration/install-community/)

{{end}}

## Creating a New Solution

In this tutorial, we will use the [ABP CLI](../../../CLI.md) to create the sample application with the ABP Framework. You can run the following command in a command-line terminal to install the **\*\*ABP CLI\*\***, if you haven't installed it yet:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then create an empty folder, open a command-line terminal and execute the following command in the terminal:

````bash

abp new TodoApp -t app-nolayers{{if UI=="BlazorServer"}} -u blazor-server{{else if UI=="Blazor"}} -u blazor{{else if UI=="NG"}} -u angular{{end}}{{if DB=="Mongo"}} -d mongodb{{end}}

````

{{if UI=="NG"}}

This will create a new solution, named *\*TodoApp\**, with `angular` and `aspnet-core` folders. Once the solution is ready, open the solution (in the `aspnet-core` folder) with your favorite IDE.

{{else if UI=="Blazor"}}

This will create a new solution with three projects:

\* A `blazor` application that contains the Blazor code, the client-side.

\* A `host` application, hosts and serves the `blazor` application.

\* A `contracts` project, shared library between these two projects.

Once the solution is ready, open it in your favorite IDE.

{{else}}

This will create a new solution with a single project, named *\*TodoApp\**. Once the solution is ready, open it in your favorite IDE.

{{end}}

### Create the Database

You can run the following command in the {{if UI=="Blazor"}} directory of your `TodoApp.Host` project {{else}}root directory of your project (in the same folder of the `.csproj` file){{end}} to create the database and seed the initial data:

```bash

dotnet run --migrate-database

```

This command will create the database and seed the initial data for you. Then you can run the application.

### Run the Application

{{if UI=="MVC" || UI=="BlazorServer"}}

It is good to run the application before starting the development. Running the application is pretty straight-forward, you can run the application with any IDE that supports .NET or by running the `dotnet run` CLI command in the directory of your project:

{{else if UI=="Blazor"}}

It is good to run the application before starting the development. Running the application is pretty straight-forward, you just need to run the `TodoApp.Host` application with any IDE that supports .NET or by running the `dotnet run` CLI command in the directory of your project.

> **\*\*Note:\*\*** The `host` application hosts and serves the `blazor` application. Therefore, you should run the `host` application only.

After the application runs, open the application in your default browser:

{{else if UI=="NG"}}

It is good to run the application before starting the development. The solution has two main applications:

\* `TodoApp` (in the .NET solution) hosts the server-side HTTP API, so the Angular application can consume it. (server-side application)

\* `angular` folder contains the Angular application. (client-side application)

Firstly, run the `TodoApp` project in your favorite IDE (or run the `dotnet run` CLI command on your project directory) to see the server-side HTTP API on [Swagger UI](https://swagger.io/tools/swagger-ui/):

![todo-swagger-ui-initial](./todo-single-layer-ui-initial.png)

You can explore and test your HTTP API with this UI. If it works, then we can run the Angular client application.

You can run the application using the following (or `yarn start`) command:

````bash

npm start

````

This command takes time, but eventually runs and opens the application in your default browser:

{{end}}

![todo-ui-initial](../todo-ui-initial.png)

You can click on the *\*Login\** button and use `admin` as the username and `1q2w3E\*` as the password to login to the application.

All right. We can start coding!

## Defining the Entity

This application will have a single [entity](../../../Entities.md) and we can start by creating it. So, create a new `TodoItem` class under the `Entities` folder of {{if UI=="Blazor"}}the `TodoApp.Host` project{{else}}the project{{end}}:

````csharp

using Volo.Abp.Domain.Entities;

namespace TodoApp{{if UI=="Blazor"}}.{{end}}Entities;

public class TodoItem : BasicAggregateRoot<Guid>

{

public string Text { get; set; }

}

````

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

## Database Integration

{{if DB=="EF"}}

Next step is to setup the [Entity Framework Core](../../../Entity-Framework-Core.md) configuration.

### Mapping Configuration

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

````csharp

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

````

Then navigate to the `OnModelCreating` method in the same class and add the following mapping code for the `TodoItem ` entity:

````csharp

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");

});

}

````

We've mapped the `TodoItem` entity to the `TodoItems` table in the database. The next step is to create a migration and apply the changes to the database.

### Code First Migrations

The startup solution is configured to use Entity Framework Core [Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations). Since we've changed the database mapping configuration, we should create a new migration and apply changes to the database.

Open a command-line terminal in the {{if UI=="Blazor"}} directory of your `TodoApp.Host` project {{else}}root directory of your project (in the same folder of the `.csproj` file){{end}} and type the following command:

````bash

dotnet ef migrations add Added\_TodoItem

````

This will add a new migration class to the project. You should see the new migration in the `Migrations` folder:

![todo-efcore-migration](todo-efcore-migration-single-layer.png)

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

````bash

dotnet ef database update

````

{{else if DB=="Mongo"}}

The next step is to setup the [MongoDB](../../../MongoDB.md) configuration. Open the `TodoAppDbContext` class (under the **\*\*Data\*\*** folder) in your project and make the following changes:

1. Add a new property to the class:

````csharp

public IMongoCollection<TodoItem> TodoItems => Collection<TodoItem>();

````

2. Add the following code inside the `CreateModel` method:

````csharp

modelBuilder.Entity<TodoItem>(b =>

{

b.CollectionName = "TodoItems";

});

````

{{end}}

After the database integrations, now we can start to create application service methods and implement our use-cases.

## Creating the Application Service

An [application service](../../../Application-Services.md) 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

Before starting to implement these use cases, first we need to create a DTO class that will be used in the application service.

### Creating the Data Transfer Object (DTO)

[Application services](../../../Application-Services.md) typically get and return DTOs ([Data Transfer Objects](../../../Data-Transfer-Objects.md)) instead of entities. So, create a new `TodoItemDto` class under the `Services/Dtos` folder{{if UI=="Blazor"}} of your `TodoApp.Contracts` project{{end}}:

```csharp

namespace TodoApp.Services.Dtos;

public class TodoItemDto

{

public Guid Id { get; set; }

public string Text { get; set; }

}

```

This is a very simple DTO class that has the same properties as the `TodoItem` entity. Now, we are ready to implement our use-cases.

{{if UI=="Blazor"}}

### The Application Service Interface

Create a `ITodoAppService` interface under the `Services` folder of the `TodoApp.Contracts` project, as shown below:

```csharp

using TodoApp.Services.Dtos;

using Volo.Abp.Application.Services;

namespace TodoApp.Services;

public interface ITodoAppService : IApplicationService

{

Task<List<TodoItemDto>> GetListAsync();

Task<TodoItemDto> CreateAsync(string text);

Task DeleteAsync(Guid id);

}

```

{{end}}

### The Application Service Implementation

Create a `TodoAppService` class under the `Services` folder of {{if UI=="Blazor"}}your `TodoApp.Host` project{{else}}your project{{end}}, as shown below:

```csharp

{{if UI=="Blazor"}}

using TodoApp.Services;

using TodoApp.Services.Dtos;

using TodoApp.Entities;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

{{else}}

using TodoApp.Entities;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

{{end}}

namespace TodoApp.Services;

public class TodoAppService : ApplicationService{{if UI=="Blazor"}}, ITodoAppService{{end}}

{

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 our use-cases. ABP provides default generic [repositories](../../../Repositories.md) for the entities. We can use them to perform the fundamental database operations. This class [injects](../../../Dependency-Injection.md) `IRepository<TodoItem, Guid>`, which is the default repository for the `TodoItem` entity. We will use it to implement our use cases.

#### Getting the Todo Items

Let's start by implementing the `GetListAsync` method, which is used to get a list of todo items:

````csharp

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.

#### Creating a New Todo Item

The next method is `CreateAsync` and we can implement it as shown below:

````csharp

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.

#### Deleting a Todo Item

Finally, we can implement the `DeleteAsync` as the following code block:

````csharp

public async Task DeleteAsync(Guid id)

{

await \_todoItemRepository.DeleteAsync(id);

}

````

The application service is ready to be used from the UI layer. So, let's implement it.

## User Interface

It is time to show the todo items on the UI! Before starting to write the code, it would be good to remember what we are trying to build. Here's a sample screenshot from the final UI:

![todo-list](../todo-list.png)

{{if UI=="MVC"}}

### Index.cshtml.cs

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

```csharp

using TodoApp.Services;

using TodoApp.Services.Dtos;

using Volo.Abp.AspNetCore.Mvc.UI.RazorPages;

namespace TodoApp.Pages;

public class IndexModel : AbpPageModel

{

public List<TodoItemDto> TodoItems { get; set; }

private readonly TodoAppService \_todoAppService;

public IndexModel(TodoAppService todoAppService)

{

\_todoAppService = todoAppService;

}

public async Task OnGetAsync()

{

TodoItems = await \_todoAppService.GetListAsync();

}

}

```

This class uses `TodoAppService` to get the list of todo items and assign the `TodoItems` property. We will use it to render the todo items on the razor page.

### Index.cshtml

Open the `Index.cshtml` file in the `Pages` folder and replace it with the following content:

```xml

@page

@model TodoApp.Pages.IndexModel

@section styles {

<abp-style src="/Pages/Index.cshtml.css" />

}

@section scripts {

<abp-script src="/Pages/Index.cshtml.js" />

}

<div class="container">

<abp-card>

<abp-card-header>

<abp-card-title>

TODO LIST

</abp-card-title>

</abp-card-header>

<abp-card-body>

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

<form id="NewItemForm" class="row row-cols-lg-auto g-3 align-items-center">

<div class="col-12">

<div class="input-group">

<input id="NewItemText" type="text" 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 Model.TodoItems)

{

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

<i class="fa fa-trash-o"></i> @todoItem.Text

</li>

}

</ul>

</abp-card-body>

</abp-card>

</div>

```

We are using ABP's [card tag helper](../../../UI/AspNetCore/Tag-Helpers/Cards.md) to create a simple card view. You could directly use the standard bootstrap HTML structure, however the ABP [tag helpers](../../../UI/AspNetCore/Tag-Helpers/Index.md) make it much easier and type safe.

This page imports a CSS and a JavaScript file, so we should also create them.

### Index.cshtml.js

Open the `Index.cshtml.js` file in the `Pages` folder and replace with the following content:

````js

$(function () {

// DELETING ITEMS /////////////////////////////////////////

$('#TodoList').on('click', 'li i', function(){

var $li = $(this).parent();

var id = $li.attr('data-id');

todoApp.services.todo.delete(id).then(function(){

$li.remove();

abp.notify.info('Deleted the todo item.');

});

});

// CREATING NEW ITEMS /////////////////////////////////////

$('#NewItemForm').submit(function(e){

e.preventDefault();

var todoText = $('#NewItemText').val();

todoApp.services.todo.create(todoText).then(function(result){

$('<li data-id="' + result.id + '">')

.html('<i class="fa fa-trash-o"></i> ' + result.text)

.appendTo($('#TodoList'));

$('#NewItemText').val('');

});

});

});

````

In the first part, we subscribed to the click events of the trash icons near the todo items, deleted the related item on the server and showed a notification on the UI. Also, we removed the deleted item from the DOM, so we wouldn't need to refresh the page.

In the second part, we created a new todo item on the server. If it succeeded, we would then manipulate the DOM to insert a new `<li>` element to the todo list. This way, we wouldn't need to refresh the whole page after creating a new todo item.

The interesting part here is how we communicate with the server. See the *\*Dynamic JavaScript Proxies & Auto API Controllers\** section to understand how it works. But now, let's continue and complete the application.

### Index.cshtml.css

As for the final touch, open the `Index.cshtml.css` file in the `Pages` folder and replace with the following content:

````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;

}

````

This is a simple styling for the todo page. We believe that you can do much better :)

Now, you can run the application again and see the result.

### Dynamic JavaScript Proxies & Auto API Controllers

In the `Index.cshtml.js` file, we've used the `todoApp.services.todo.delete(...)` and `todoApp.services.todo.create(...)` functions to communicate with the server. These functions are dynamically created by the ABP Framework, thanks to the [Dynamic JavaScript Client Proxy](../../../UI/AspNetCore/Dynamic-JavaScript-Proxies.md) system. They perform HTTP API calls to the server and return a promise, so you can register a callback to the `then` function as we've done above.

> `services` keyword comes from the namespace (`namespace TodoApp.Services;`). It's a naming convention.

However, you may notice that we haven't created any API Controllers, so how does the server handle these requests? This question brings us to the [Auto API Controller](../../../API/Auto-API-Controllers.md) feature of the ABP Framework. It automatically converts the application services to **\*\*API Controllers\*\*** by convention.

If you open [Swagger UI](https://swagger.io/tools/swagger-ui/) by entering the `/swagger` URL in your application, you can see the Todo API:

![todo-api](../todo-api.png)

{{else if UI=="Blazor" || UI=="BlazorServer"}}

### Index.razor.cs

Open the `Index.razor.cs` file in the `Pages` folder{{if UI=="Blazor"}} in your `Todo.Blazor` project{{end}} and replace the content with the following code block:

```csharp

{{if UI=="Blazor"}}

using Microsoft.AspNetCore.Components;

using TodoApp.Services;

using TodoApp.Services.Dtos;

{{else}}

using Microsoft.AspNetCore.Components;

using TodoApp.Services;

using TodoApp.Services.Dtos;

{{end}}

namespace TodoApp.Pages;

public partial class Index

{

[Inject]

private {{if UI=="Blazor"}}ITodoAppService{{else}}TodoAppService{{end}} 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 the {{if UI=="Blazor"}}`ITodoAppService`{{else}}`TodoAppService`{{end}} to get the list of 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.

### Index.razor

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

```xml

@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>

```

### Index.razor.css

As the final touch, open the `Index.razor.css` file in the `Pages` folder and replace it with the following content:

````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;

}

````

This is a simple styling for the todo page. We believe that you can do much better :)

Now, you can run the {{if UI=="Blazor"}}`TodoApp.Host` project{{else}}application{{end}} again to see the result.

{{else if UI=="NG"}}

### Service Proxy Generation

ABP provides a handy feature to automatically create client-side services to easily consume HTTP APIs provided by the server.

You first need to run the `TodoApp` project since the proxy generator reads API definitions from the server application.

Once you run the `TodoApp` project (**\*\*Swagger API Definition\*\*** will be shown), open a command-line terminal in the directory of `angular` folder and run the following command:

```bash

abp generate-proxy -t ng

```

If everything goes well, it should generate an output as shown below:

```bash

CREATE src/app/proxy/generate-proxy.json (182755 bytes)

CREATE src/app/proxy/README.md (1000 bytes)

CREATE src/app/proxy/services/todo.service.ts (833 bytes)

CREATE src/app/proxy/services/dtos/models.ts (71 bytes)

CREATE src/app/proxy/services/dtos/index.ts (26 bytes)

CREATE src/app/proxy/services/index.ts (81 bytes)

CREATE src/app/proxy/index.ts (61 bytes)

```

Then, we can use the `TodoService` to use the server-side HTTP APIs, as we'll do in the next section.

### home.component.ts

Open the `/angular/src/app/home/home.component.ts` file and replace its content with the following code block:

```ts

import { ToasterService } from "@abp/ng.theme.shared";

import { Component, OnInit } from '@angular/core';

import { TodoItemDto } from "@proxy/services/dtos";

import { TodoService } from "@proxy/services";

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.scss'],

})

export class HomeComponent implements OnInit {

todoItems: TodoItemDto[];

newTodoText: string;

constructor(

private todoService: TodoService,

private toasterService: ToasterService)

{ }

ngOnInit(): void {

this.todoService.getList().subscribe(response => {

this.todoItems = response;

});

}

create(): void{

this.todoService.create(this.newTodoText).subscribe((result) => {

this.todoItems = this.todoItems.concat(result);

this.newTodoText = null;

});

}

delete(id: string): void {

this.todoService.delete(id).subscribe(() => {

this.todoItems = this.todoItems.filter(item => item.id !== id);

this.toasterService.info('Deleted the todo item.');

});

}

}

```

We've used `TodoService` to get the list of todo items and assigned the returning value to the `todoItems` array. We've also added `create` and `delete` methods. These methods will be used on the view side.

### home.component.html

Open the `/angular/src/app/home/home.component.html` file and replace its content with the following code block:

````html

<div class="container">

<div class="card">

<div class="card-header">

<div class="card-title">TODO LIST</div>

</div>

<div class="card-body">

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

<form class="row row-cols-lg-auto g-3 align-items-center" (ngSubmit)="create()">

<div class="col-12">

<div class="input-group">

<input name="NewTodoText" type="text" [(ngModel)]="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">

<li \*ngFor="let todoItem of todoItems">

<i class="fa fa-trash-o" (click)="delete(todoItem.id)"></i> {%{{{ todoItem.text }}}%}

</li>

</ul>

</div>

</div>

</div>

````

### home.component.scss

As the final touch, open the `/angular/src/app/home/home.component.scss` file and replace its content with the following code block:

````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;

}

````

This is a simple styling for the todo page. We believe that you can do much better :)

Now, you can run the application again to see the result.

{{end}}

## Conclusion

In this tutorial, we've built a very simple application to warm up with the ABP Framework.

## Source Code

You can find the source code of the completed application [here](https://github.com/abpframework/abp-samples/tree/master/TodoApp-SingleLayer).

## See Also

\* Check the [Web Application Development Tutorial](../../Part-1.md) to see a real-life web application development in a layered architecture using the [Application Startup Template](../../../Startup-Templates/Application.md).

## With layered architecture

# Quick Start

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

This is a single-part quick-start tutorial to build a simple todo application with the ABP Framework. Here's a screenshot from the final application:

![todo-list](todo-list.png)

You can find the source code of the completed application [here](https://github.com/abpframework/abp-samples/tree/master/TodoApp).

This documentation has a video tutorial on **\*\*YouTube\*\***!! You can watch it here:

{{if UI=="MVC" && DB =="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/763DV0fwSbk" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="Blazor" && DB=="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/ivxJsi8c7-8" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="BlazorServer" && DB=="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/1BdYg5NLrJs" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="NG" && DB=="EF"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/Lqh1j1H5pkg" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="MVC" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/7Rm-K2re4MI" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="BlazorServer" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/i23C8hN7OAs" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="Blazor" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/JpiMiXOBG6A" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{else if UI=="NG" && DB=="Mongo"}}

<iframe width="560" height="315" src="https://www.youtube.com/embed/DOxk9Doxad0" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{{end}}

## Pre-Requirements

\* An IDE (e.g. [Visual Studio](https://visualstudio.microsoft.com/vs/)) that supports [.NET 7.0+](https://dotnet.microsoft.com/download/dotnet) development.

\* [Node v16.x](https://nodejs.org/)

{{if DB=="Mongo"}}

\* [MongoDB Server 4.0+](https://docs.mongodb.com/manual/administration/install-community/)

{{end}}

## Install ABP CLI Tool

We will use the [ABP CLI](../../CLI.md) to create new ABP solutions. You can run the following command on a terminal window to install this dotnet tool:

````bash

dotnet tool install -g Volo.Abp.Cli

````

## Create Your ABP Solution

Create an empty folder, open a command-line terminal and execute the following command in the terminal:

````bash

abp new TodoApp{{if UI=="Blazor"}} -u blazor{{else if UI=="BlazorServer"}} -u blazor-server{{else if UI=="NG"}} -u angular{{end}}{{if DB=="Mongo"}} -d mongodb{{end}}

````

{{if UI=="NG"}}

This will create a new solution, named *\*TodoApp\** with `angular` and `aspnet-core` folders. Once the solution is ready, open the ASP.NET Core solution in your favorite IDE.

{{else}}

This will create a new solution, named *\*TodoApp\**. Once the solution is ready, open it in your favorite IDE.

{{end}}

### 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.

{{if DB=="EF"}}

> 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.

{{end}}

### Run the Application

{{if UI=="MVC" || UI=="BlazorServer"}}

It is good to run the application before starting the development. Ensure the {{if UI=="BlazorServer"}}`TodoApp.Blazor`{{else}}`TodoApp.Web`{{end}} project is the startup project, then run the application (Ctrl+F5 in Visual Studio) to see the initial UI:

{{else if UI=="Blazor"}}

It is good to run the application before starting the development. The solution has two main applications;

\* `TodoApp.HttpApi.Host` hosts the server-side HTTP API.

\* `TodoApp.Blazor` is the client-side Blazor WebAssembly application.

Ensure the `TodoApp.HttpApi.Host` project is the startup project, then run the application (Ctrl+F5 in Visual Studio) to see the server-side HTTP API on the [Swagger UI](https://swagger.io/tools/swagger-ui/):

![todo-swagger-ui-initial](todo-swagger-ui-initial.png)

You can explore and test your HTTP API with this UI. Now, we can set the `TodoApp.Blazor` as the startup project and run it to open the actual Blazor application UI:

{{else if UI=="NG"}}

It is good to run the application before starting the development. The solution has two main applications:

\* `TodoApp.HttpApi.Host` (in the .NET solution) host the server-side HTTP API.

\* `angular` folder contains the Angular application.

Ensure that the `TodoApp.HttpApi.Host` project is the startup project, then run the application (Ctrl+F5 in Visual Studio) to see the server-side HTTP API on the [Swagger UI](https://swagger.io/tools/swagger-ui/):

![todo-swagger-ui-initial](todo-swagger-ui-initial.png)

You can explore and test your HTTP API with this UI. If it works, we can run the Angular client application.

You can run the application using the following command:

````bash

npm start

````

This command takes time, but eventually runs and opens the application in your default browser:

{{end}}

![todo-ui-initial](todo-ui-initial.png)

You can click on the *\*Login\** button, use `admin` as the username and `1q2w3E\*` as the password to login to the application.

All ready. We can start coding!

## Domain Layer

This application has a single [entity](../../Entities.md) and we'll start by creating it. Create a new `TodoItem` class inside the *\*TodoApp.Domain\** project:

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace TodoApp

{

public class TodoItem : BasicAggregateRoot<Guid>

{

public string Text { get; set; }

}

}

````

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

## Database Integration

{{if DB=="EF"}}

Next step is to setup the [Entity Framework Core](../../Entity-Framework-Core.md) configuration.

### Mapping Configuration

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

````csharp

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

````

Then navigate to the `OnModelCreating` method in the `TodoAppDbContext` class and add the mapping code for the `TodoItem ` entity:

````csharp

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");

});

}

````

We've mapped the `TodoItem` entity to the `TodoItems` table in the database.

### Code First Migrations

The startup solution is configured to use Entity Framework Core [Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations). Since we've changed the database mapping configuration, we should create a new migration and apply changes to the database.

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

````bash

dotnet ef migrations add Added\_TodoItem

````

This will add a new migration class to the project:

![todo-efcore-migration](todo-efcore-migration.png)

You can apply changes to the database using the following command, in the same command-line terminal:

````bash

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 {{if UI=="MVC"}}`TodoApp.Web`{{else if UI=="BlazorServer"}}`TodoApp.Blazor`{{else if UI=="Blazor" || UI=="NG"}}`TodoApp.HttpApi.Host`{{end}} is the startup project and `TodoApp.EntityFrameworkCore` is the *\*Default Project\** in PMC.

{{else if DB=="Mongo"}}

Next step is to setup the [MongoDB](../../MongoDB.md) configuration. Open the `TodoAppMongoDbContext` class in the `MongoDb` folder of the *\*TodoApp.MongoDB\** project and make the following changes:

1. Add a new property to the class:

````csharp

public IMongoCollection<TodoItem> TodoItems => Collection<TodoItem>();

````

2. Add the following code inside the `CreateModel` method:

````csharp

modelBuilder.Entity<TodoItem>(b =>

{

b.CollectionName = "TodoItems";

});

````

{{end}}

Now, we can use the ABP repositories to save and retrieve the todo items, as we'll do in the next section.

## Application Layer

An [Application Service](../../Application-Services.md) 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

### 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:

````csharp

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);

}

}

````

### Data Transfer Object

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

````csharp

using System;

namespace TodoApp

{

public class TodoItemDto

{

public Guid Id { get; set; }

public string Text { get; set; }

}

}

````

This is a very simple DTO class that matches our `TodoItem` entity. We are ready to implement the `ITodoAppService`.

### Application Service Implementation

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

````csharp

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](../../Repositories.md) for the entities. We can use them to perform the fundamental database operations. This class [injects](../../Dependency-Injection.md) `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

Let's start by implementing the `GetListAsync` method:

````csharp

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 complete `TodoItem` list from the database, mapping them to `TodoItemDto` objects and returning as the result.

#### Creating a New Todo Item

Next method is `CreateAsync` and we can implement it as shown below:

````csharp

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.

#### Deleting a Todo Item

Finally, we can implement the `DeleteAsync` as the following code block:

````csharp

public async Task DeleteAsync(Guid id)

{

await \_todoItemRepository.DeleteAsync(id);

}

````

The application service is ready to be used from the UI layer.

## User Interface Layer

It is time to show the todo items on the UI! Before starting to write the code, it would be good to remember what we are trying to build. Here's a sample screenshot from the final UI:

![todo-list](todo-list.png)

> **\*\*We will keep the UI side minimal for this tutorial to make the tutorial simple and focused. See the [**web application development tutorial**](../Part-1.md) to build real-life pages with all aspects.\*\***

{{if UI=="MVC"}}

### Index.cshtml.cs

Open the `Index.cshtml.cs` file in the `Pages` folder of the *\*TodoApp.Web\** project and replace the content with the following code block:

````csharp

using System.Collections.Generic;

using System.Threading.Tasks;

namespace TodoApp.Web.Pages

{

public class IndexModel : TodoAppPageModel

{

public List<TodoItemDto> TodoItems { get; set; }

private readonly ITodoAppService \_todoAppService;

public IndexModel(ITodoAppService todoAppService)

{

\_todoAppService = todoAppService;

}

public async Task OnGetAsync()

{

TodoItems = await \_todoAppService.GetListAsync();

}

}

}

````

This class uses the `ITodoAppService` to get the list of todo items and assign the `TodoItems` property. We will use it to render the todo items on the razor page.

### Index.cshtml

Open the `Index.cshtml` file in the `Pages` folder of the *\*TodoApp.Web\** project and replace it with the following content:

````xml

@page

@model TodoApp.Web.Pages.IndexModel

@section styles {

<abp-style src="/Pages/Index.css" />

}

@section scripts {

<abp-script src="/Pages/Index.js" />

}

<div class="container">

<abp-card>

<abp-card-header>

<abp-card-title>

TODO LIST

</abp-card-title>

</abp-card-header>

<abp-card-body>

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

<form id="NewItemForm" class="row row-cols-lg-auto g-3 align-items-center">

<div class="col-12">

<div class="input-group">

<input id="NewItemText" type="text" 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 Model.TodoItems)

{

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

<i class="fa fa-trash-o"></i> @todoItem.Text

</li>

}

</ul>

</abp-card-body>

</abp-card>

</div>

````

We are using ABP's [card tag helper](../../UI/AspNetCore/Tag-Helpers/Cards.md) to create a simple card view. You could directly use the standard bootstrap HTML structure, however the ABP [tag helpers](../../UI/AspNetCore/Tag-Helpers/Index.md) make it much easier and type safe.

This page imports a CSS and a JavaScript file, so we should also create them.

### Index.js

Open the `Index.js` file in the `Pages` folder of the *\*TodoApp.Web\** project and replace it with the following content:

````js

$(function () {

// DELETING ITEMS /////////////////////////////////////////

$('#TodoList').on('click', 'li i', function(){

var $li = $(this).parent();

var id = $li.attr('data-id');

todoApp.todo.delete(id).then(function(){

$li.remove();

abp.notify.info('Deleted the todo item.');

});

});

// CREATING NEW ITEMS /////////////////////////////////////

$('#NewItemForm').submit(function(e){

e.preventDefault();

var todoText = $('#NewItemText').val();

todoApp.todo.create(todoText).then(function(result){

$('<li data-id="' + result.id + '">')

.html('<i class="fa fa-trash-o"></i> ' + result.text)

.appendTo($('#TodoList'));

$('#NewItemText').val('');

});

});

});

````

In the first part, we are subscribing to the click events of the trash icons near the todo items, deleting the related item on the server and showing a notification on the UI. Also, we are removing the deleted item from the DOM, so we don't need to refresh the page.

In the second part, we are creating a new todo item on the server. If it succeeds, we are then manipulating the DOM to insert a new `<li>` element to the todo list. This way we don't need to refresh the whole page after creating a new todo item.

The interesting part here is how we communicate with the server. See the *\*Dynamic JavaScript Proxies & Auto API Controllers\** section to understand how it works. But now, let's continue and complete the application.

### Index.css

As the final touch, open the `Index.css` file in the `Pages` folder of the *\*TodoApp.Web\** project and replace it with the following content:

````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;

}

````

This is a simple styling for the todo page. We believe that you can do much better :)

Now, you can run the application again and see the result.

### Dynamic JavaScript Proxies & Auto API Controllers

In the `Index.js` file, we've used the `todoApp.todo.delete(...)` and `todoApp.todo.create(...)` functions to communicate with the server. These functions are dynamically created by the ABP Framework, thanks to the [Dynamic JavaScript Client Proxy](../../UI/AspNetCore/Dynamic-JavaScript-Proxies.md) system. They perform HTTP API calls to the server and return a promise, so you can register a callback to the `then` function as we've done above.

However, you may notice that we haven't created any API Controllers, so how does the server handle these requests? This question brings us to the [Auto API Controller](../../API/Auto-API-Controllers.md) feature of the ABP Framework. It automatically converts the application services to API Controllers by convention.

If you open the [Swagger UI](https://swagger.io/tools/swagger-ui/) by entering the `/swagger` URL in your application, you can see the Todo API:

![todo-api](todo-api.png)

{{else if UI=="Blazor" || UI=="BlazorServer"}}

### Index.razor.cs

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

````csharp

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.

{{if UI=="Blazor"}}

See the *\*Dynamic C# Proxies & Auto API Controllers\** section below to learn how we could inject and use the application service interface from the Blazor application which is running on the browser! But now, let's continue and complete the application.

{{end # Blazor}}

### Index.razor

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

````xml

@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>

````

### Index.razor.css

As the final touch, open the `Index.razor.css` file in the `Pages` folder of the *\*TodoApp.Blazor\** project and replace it with the following content:

````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;

}

````

This is a simple styling for the todo page. We believe that you can do much better :)

Now, you can run the application again to see the result.

{{if UI=="Blazor"}}

### Dynamic C# Proxies & Auto API Controllers

In the `Index.razor.cs` file, we've injected (with the `[Inject]` attribute) and used the `ITodoAppService` just like using a local service. Remember that the Blazor application is running on the browser while the implementation of this application service is running on the server.

The magic is done by the ABP Framework's [Dynamic C# Client Proxy](../../API/Dynamic-CSharp-API-Clients.md) system. It uses the standard `HttpClient` and performs HTTP API requests to the remote server. It also handles all the standard tasks for us, including authorization, JSON serialization and exception handling.

However, you may ask that we haven't created any API Controller, so how does the server handle these requests? This question brings us to the [Auto API Controller](../../API/Auto-API-Controllers.md) feature of the ABP Framework. It automatically converts the application services to API Controllers by convention.

If you run the `TodoApp.HttpApi.Host` application, you can see the Todo API:

![todo-api](todo-api.png)

{{end # Blazor}}

{{else if UI=="NG"}}

### Service Proxy Generation

ABP provides a handy feature to automatically create client-side services to easily consume HTTP APIs provided by the server.

You first need to run the `TodoApp.HttpApi.Host` project since the proxy generator reads API definitions from the server application.

> **\*\*Warning\*\***: There is an issue with IIS Express: it doesn't allow connecting to the application from another process. If you are using Visual Studio, select the `TodoApp.HttpApi.Host` instead of IIS Express in the run button drop-down list, as shown in the figure below:

![run-without-iisexpress](run-without-iisexpress.png)

Once you run the `TodoApp.HttpApi.Host` project, open a command-line terminal in the `angular` folder and type the following command:

````bash

abp generate-proxy -t ng

````

If everything goes well, it should generate an output as shown below:

````bash

CREATE src/app/proxy/generate-proxy.json (170978 bytes)

CREATE src/app/proxy/README.md (1000 bytes)

CREATE src/app/proxy/todo.service.ts (794 bytes)

CREATE src/app/proxy/models.ts (66 bytes)

CREATE src/app/proxy/index.ts (58 bytes)

````

We can then use `todoService` to use the server-side HTTP APIs, as we'll do in the next section.

### home.component.ts

Open the `/angular/src/app/home/home.component.ts` file and replace its content with the following code block:

````js

import { ToasterService } from '@abp/ng.theme.shared';

import { Component, OnInit } from '@angular/core';

import { TodoItemDto, TodoService } from '@proxy';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.scss']

})

export class HomeComponent implements OnInit {

todoItems: TodoItemDto[];

newTodoText: string;

constructor(

private todoService: TodoService,

private toasterService: ToasterService)

{ }

ngOnInit(): void {

this.todoService.getList().subscribe(response => {

this.todoItems = response;

});

}

create(): void{

this.todoService.create(this.newTodoText).subscribe((result) => {

this.todoItems = this.todoItems.concat(result);

this.newTodoText = null;

});

}

delete(id: string): void {

this.todoService.delete(id).subscribe(() => {

this.todoItems = this.todoItems.filter(item => item.id !== id);

this.toasterService.info('Deleted the todo item.');

});

}

}

````

We've used `todoService` to get the list of todo items and assigned the returning value to the `todoItems` array. We've also added `create` and `delete` methods. These methods will be used on the view side.

### home.component.html

Open the `/angular/src/app/home/home.component.html` file and replace its content with the following code block:

````html

<div class="container">

<div class="card">

<div class="card-header">

<div class="card-title">TODO LIST</div>

</div>

<div class="card-body">

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

<form class="row row-cols-lg-auto g-3 align-items-center" (ngSubmit)="create()">

<div class="col-12">

<div class="input-group">

<input name="NewTodoText" type="text" [(ngModel)]="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">

<li \*ngFor="let todoItem of todoItems">

<i class="fa fa-trash-o" (click)="delete(todoItem.id)"></i> {%{{{ todoItem.text }}}%}

</li>

</ul>

</div>

</div>

</div>

````

### home.component.scss

As the final touch, open the `/angular/src/app/home/home.component.scss` file and replace its content with the following code block:

````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;

}

````

This is a simple styling for the todo page. We believe that you can do much better :)

Now, you can run the application again to see the result.

{{end}}

## Conclusion

In this tutorial, we've built a very simple application to warm up for the ABP Framework. If you are looking to build a serious application, please check the [web application development tutorial](../Part-1.md) which covers all the aspects of real-life web application development.

## Source Code

You can find source code of the completed application [here](https://github.com/abpframework/abp-samples/tree/master/TodoApp).

## See Also

\* [Web Application Development Tutorial](../Part-1.md)

# Getting Started

# Getting Started: Overall

## Select the Solution Architecture

This tutorial has multiple versions. Please select the one that fits you the best:

\* **\*\*[**Single-Layer Solution**](Getting-Started-Single-Layered.md)\*\***: Creates a single-project solution. Recommended for building an application with a **\*\*simpler and easy to understand\*\*** architecture.

\* **\*\*[**Layered Solution Architecture**](Getting-Started.md)\*\***: A fully layered (multiple projects) solution based on [Domain Driven Design](Domain-Driven-Design.md) practices. Recommended for long-term projects that need a **\*\*maintainable and extensible\*\*** codebase.

## Web Application - Layered Architecture

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"],

"Tiered": ["Yes", "No"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

This tutorial explains how to **\*\*create and run\*\*** a new web application using the ABP Framework. Follow the steps below;

1. [Setup your development environment](Getting-Started-Setup-Environment.md)

2. [Creating a new solution](Getting-Started-Create-Solution.md)

3. [Running the solution](Getting-Started-Running-Solution.md)

### 1: Setup Your Development Environment

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"],

"Tiered": ["Yes", "No"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

## Setup Your Development Environment

First things first! Let's setup your development environment before creating the project.

### Pre-Requirements

The following tools should be installed on your development machine:

\* An IDE (e.g. [Visual Studio](https://visualstudio.microsoft.com/vs/)) that supports [.NET 7.0+](https://dotnet.microsoft.com/download/dotnet) development.

{{ if UI != "Blazor" }}

\* [Node v16 or v18](https://nodejs.org/)

\* [Yarn v1.20+ (not v2)](https://classic.yarnpkg.com/en/docs/install) <sup id="a-yarn">[1](#f-yarn)</sup> or npm v6+ (already installed with Node)

{{ end }}

{{ if Tiered == "Yes" }}

\* [Redis](https://redis.io/) (the startup solution uses the Redis as the [distributed cache](Caching.md)).

{{ end }}

{{ if UI != "Blazor" }}

<sup id="f-yarn"><b>1</b></sup> *\_Yarn v2 works differently and is not supported.\_* <sup>[↩](#a-yarn)</sup>

{{ end }}

### Install the ABP CLI

[ABP CLI](./CLI.md) is a command line interface that is used to automate some common tasks for ABP based solutions. First, you need to install the ABP CLI using the following command:

````shell

dotnet tool install -g Volo.Abp.Cli

````

If you've already installed, you can update it using the following command:

````shell

dotnet tool update -g Volo.Abp.Cli

````

## Next Step

\* [Creating a new solution](Getting-Started-Create-Solution.md)

### 2: Creating a New Solution

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"],

"Tiered": ["Yes", "No"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

## Create a New Project

We will use the ABP CLI to create a new ABP project.

> Alternatively, you can **\*\*create and download\*\*** projects from the [ABP Framework website](https://abp.io/get-started) by easily selecting all options from the page.

Use the `new` command of the ABP CLI to create a new project:

````shell

abp new Acme.BookStore{{if UI == "NG"}} -u angular{{else if UI == "Blazor"}} -u blazor{{else if UI == "BlazorServer"}} -u blazor-server{{end}}{{if DB == "Mongo"}} -d mongodb{{end}}{{if Tiered == "Yes"}}{{if UI == "MVC" || UI == "BlazorServer"}} --tiered{{else}} --separate-auth-server{{end}}{{end}}

````

*\*You can use different level of namespaces; e.g. BookStore, Acme.BookStore or Acme.Retail.BookStore.\**

{{ if Tiered == "Yes" }}

{{ if UI == "MVC" || UI == "BlazorServer" }}

\* `--tiered` argument is used to create N-tiered solution where authentication server, UI and API layers are physically separated.

{{ else }}

\* `--separate-auth-server` argument is used to separate the Auth Server application from the API host application. If not specified, you will have a single endpoint on the server.

{{ end }}

{{ end }}

> [ABP CLI document](./CLI.md) covers all of the available commands and options.

### Mobile Development

If you want to include a [React Native](https://reactnative.dev/) project in your solution, add `-m react-native` (or `--mobile react-native`) argument to project creation command. This is a basic React Native startup template to develop mobile applications integrated to your ABP based backends.

See the [Getting Started with the React Native](Getting-Started-React-Native.md) document to learn how to configure and run the React Native application.

## The Solution Structure

The solution has a layered structure (based on the [Domain Driven Design](Domain-Driven-Design.md)) and contains unit & integration test projects. See the [application template document](Startup-Templates/Application.md) to understand the solution structure in details.

{{ if DB == "Mongo" }}

## MongoDB Transactions

The [startup template](Startup-templates/Index.md) **\*\*disables\*\*** transactions in the `.MongoDB` project by default. If your MongoDB server supports transactions, you can enable it in the *\*YourProjectMongoDbModule\** class's `ConfigureServices` method:

```csharp

Configure<AbpUnitOfWorkDefaultOptions>(options =>

{

options.TransactionBehavior = UnitOfWorkTransactionBehavior.Auto;

});

```

> Or you can delete that code since `Auto` is already the default behavior.

{{ end }}

## Next Step

\* [Running the solution](Getting-Started-Running-Solution.md)

### 3: Running the Solution

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"],

"Tiered": ["Yes", "No"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

## Create the Database

### Connection String

Check the **\*\*connection string\*\*** in the `appsettings.json` file under the {{if Tiered == "Yes"}}`.AuthServer` and `.HttpApi.Host` projects{{else}}{{if UI=="MVC"}}`.Web` project{{else if UI=="BlazorServer"}}`.Blazor` project{{else}}`.HttpApi.Host` project{{end}}{{end}}.

{{ if DB == "EF" }}

````json

"ConnectionStrings": {

"Default": "Server=(LocalDb)\MSSQLLocalDB;Database=BookStore;Trusted\_Connection=True"

}

````

> **\*\*About the Connection Strings and Database Management Systems\*\***

>

> The solution is configured to use **\*\*Entity Framework Core\*\*** with **\*\*MS SQL Server\*\*** by default. However, if you've selected another DBMS using the `-dbms` parameter on the ABP CLI `new` command (like `-dbms MySQL`), the connection string might be different for you.

>

> EF Core supports [various](https://docs.microsoft.com/en-us/ef/core/providers/) database providers and you can use any supported DBMS. See [the Entity Framework integration document](Entity-Framework-Core.md) to learn how to [switch to another DBMS](Entity-Framework-Core-Other-DBMS.md) if you need later.

### Database Migrations

The solution uses the [Entity Framework Core Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli). It comes with a `.DbMigrator` console application which **\*\*applies the migrations\*\*** and also **\*\*seeds the initial data\*\***. It is useful on **\*\*development\*\*** as well as on **\*\*production\*\*** environment.

> `.DbMigrator` project has its own `appsettings.json`. So, if you have changed the connection string above, you should also change this one.

### The Initial Migration

`.DbMigrator` application automatically **\*\*creates the Initial migration\*\*** on first run.

**\*\*If you are using Visual Studio, you can skip to the *\*Running the DbMigrator\** section.\*\*** However, other IDEs (e.g. Rider) may have problems for the first run since it adds the initial migration and compiles the project. In this case, open a command line terminal in the folder of the `.DbMigrator` project and run the following command:

````bash

dotnet run

````

For the next time, you can just run it in your IDE as you normally do.

### Running the DbMigrator

Right click to the `.DbMigrator` project and select **\*\*Set as StartUp Project\*\***

![set-as-startup-project](images/set-as-startup-project.png)

Hit F5 (or Ctrl+F5) to run the application. It will have an output like shown below:

![db-migrator-output](images/db-migrator-output.png)

> Initial [seed data](Data-Seeding.md) creates the `admin` user in the database (with the password is `1q2w3E\*`) which is then used to login to the application. So, you need to use `.DbMigrator` at least once for a new database.

{{ else if DB == "Mongo" }}

````json

"ConnectionStrings": {

"Default": "mongodb://localhost:27017/BookStore"

}

````

The solution is configured to use **\*\*MongoDB\*\*** in your local computer, so you need to have a MongoDB server instance up and running or change the connection string to another MongoDB server.

### Seed Initial Data

The solution comes with a `.DbMigrator` console application which **\*\*seeds the initial data\*\***. It is useful on **\*\*development\*\*** as well as on **\*\*production\*\*** environment.

> `.DbMigrator` project has its own `appsettings.json`. So, if you have changed the connection string above, you should also change this one.

Right click to the `.DbMigrator` project and select **\*\*Set as StartUp Project\*\***

![set-as-startup-project](images/set-as-startup-project.png)

Hit F5 (or Ctrl+F5) to run the application. It will have an output like shown below:

![db-migrator-output](images/db-migrator-output.png)

> Initial [seed data](Data-Seeding.md) creates the `admin` user in the database (with the password is `1q2w3E\*`) which is then used to login to the application. So, you need to use `.DbMigrator` at least once for a new database.

{{ end }}

## Run the Application

{{ if UI == "MVC" || UI == "BlazorServer" }}

> Before starting the application, run `abp install-libs` command in your Web directory to restore the client-side libraries. This will populate the `libs` folder.

{{ if Tiered == "Yes" }}

> Tiered solutions use **\*\*Redis\*\*** as the distributed cache. Ensure that it is installed and running in your local computer. If you are using a remote Redis Server, set the configuration in the `appsettings.json` files of the projects below.

1. Ensure that the `.AuthServer` project is the startup project. Run this application that will open a **\*\*login\*\*** page in your browser.

> Use Ctrl+F5 in Visual Studio (instead of F5) to run the application without debugging. If you don't have a debug purpose, this will be faster.

You can login, but you cannot enter to the main application here. This is **\*\*just the authentication server\*\***.

2. Ensure that the `.HttpApi.Host` project is the startup project and run the application which will open a **\*\*Swagger UI\*\*** in your browser.

![swagger-ui](images/swagger-ui.png)

This is the HTTP API that is used by the web application.

3. Lastly, ensure that the {{if UI=="MVC"}}`.Web`{{else}}`.Blazor`{{end}} project is the startup project and run the application which will open a **\*\*welcome\*\*** page in your browser

![mvc-tiered-app-home](images/bookstore-home-2.png)

Click to the **\*\*login\*\*** button which will redirect you to the *\*authentication server\** to login to the application:

![bookstore-login](images/bookstore-login-2.png)

{{ else # Tiered != "Yes" }}

Ensure that the {{if UI=="MVC"}}`.Web`{{else}}`.Blazor`{{end}} project is the startup project. Run the application which will open the **\*\*login\*\*** page in your browser:

> Use Ctrl+F5 in Visual Studio (instead of F5) to run the application without debugging. If you don't have a debug purpose, this will be faster.

![bookstore-login](images/bookstore-login-2.png)

{{ end # Tiered }}

{{ else # UI != MVC || BlazorServer }}

### Running the HTTP API Host (Server Side)

{{ if Tiered == "Yes" }}

> Tiered solutions use Redis as the distributed cache. Ensure that it is installed and running in your local computer. If you are using a remote Redis Server, set the configuration in the `appsettings.json` files of the projects below.

Ensure that the `.AuthServer` project is the startup project. Run the application which will open a **\*\*login\*\*** page in your browser.

> Use Ctrl+F5 in Visual Studio (instead of F5) to run the application without debugging. If you don't have a debug purpose, this will be faster.

You can login, but you cannot enter to the main application here. This is **\*\*just the authentication server\*\***.

Ensure that the `.HttpApi.Host` project is the startup project and run the application which will open a Swagger UI:

{{ else # Tiered == "No" }}

Ensure that the `.HttpApi.Host` project is the startup project and run the application which will open a Swagger UI:

> Use Ctrl+F5 in Visual Studio (instead of F5) to run the application without debugging. If you don't have a debug purpose, this will be faster.

{{ end # Tiered }}

![swagger-ui](images/swagger-ui.png)

You can see the application APIs and test them here. Get [more info](https://swagger.io/tools/swagger-ui/) about the Swagger UI.

{{ end # UI }}

{{ if UI == "Blazor" }}

### Running the Blazor Application (Client Side)

Go to the Blazor project folder, open a command line terminal, type the `abp bundle -f` command (If the project was created by ABP Cli tool, you don't need to do this).

Ensure that the `.Blazor` project is the startup project and run the application.

> Use Ctrl+F5 in Visual Studio (instead of F5) to run the application without debugging. If you don't have a debug purpose, this will be faster.

Once the application starts, click to the **\*\*Login\*\*** link on to header, which redirects you to the authentication server to enter a username and password:

![bookstore-login](images/bookstore-login-2.png)

{{ else if UI == "NG" }}

### Running the Angular Application (Client Side)

Go to the `angular` folder, open a command line terminal, type the `yarn` command (we suggest to the [yarn](https://yarnpkg.com/) package manager while `npm install` will also work)

```bash

yarn

```

Once all node modules are loaded, execute `yarn start` (or `npm start`) command:

```bash

yarn start

```

It may take a longer time for the first build. Once it finishes, it opens the Angular UI in your default browser with the [localhost:4200](http://localhost:4200/) address.

![bookstore-login](images/bookstore-login-2.png)

{{ end }}

Enter **\*\*admin\*\*** as the username and **\*\*1q2w3E\*\***\* as the password to login to the application. The application is up and running. You can start developing your application based on this startup template.

## See Also

\* [Web Application Development Tutorial](Tutorials/Part-1.md)

\* [Application Startup Template](Startup-Templates/Application.md)

## Web Application - Single-Layered Architecture

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

This tutorial explains how to **\*\*create and run\*\*** a new Single-Layered web application using the ABP Framework. Follow the steps below:

1. [Setup your development environment](Getting-Started-Setup-Environment-Single-Layer.md)

2. [Creating a new solution](Getting-Started-Create-Solution-Single-Layer.md)

3. [Running the solution](Getting-Started-Running-Solution-Single-Layer.md)

### 1: Setup Your Development Environment

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

## Setup Your Development Environment

First things first! Let's setup your development environment before creating the project.

### Pre-Requirements

The following tools should be installed on your development machine:

\* An IDE (e.g. [Visual Studio](https://visualstudio.microsoft.com/vs/)) that supports [.NET 7.0+](https://dotnet.microsoft.com/download/dotnet) development.

{{ if UI != "Blazor" }}

\* [Node v16 or v18](https://nodejs.org/)

\* [Yarn v1.20+ (not v2)](https://classic.yarnpkg.com/en/docs/install) <sup id="a-yarn">[1](#f-yarn)</sup> or npm v6+ (already installed with Node)

{{ end }}

{{ if UI != "Blazor" }}

<sup id="f-yarn"><b>1</b></sup> *\_Yarn v2 works differently and is not supported.\_* <sup>[↩](#a-yarn)</sup>

{{ end }}

### Install the ABP CLI

[ABP CLI](./CLI.md) is a command line interface that is used to automate some common tasks for ABP based solutions. First, you need to install the ABP CLI using the following command:

````shell

dotnet tool install -g Volo.Abp.Cli

````

If you've already installed, you can update it using the following command:

````shell

dotnet tool update -g Volo.Abp.Cli

````

## Next Step

\* [Creating a new solution](Getting-Started-Create-Solution-Single-Layer.md)

### 2: Creating a New Solution

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

## Create a New Project

We will use the ABP CLI to create a new ABP project.

> You can also use the ABP CLI Command Generator on the [ABP Framework website](https://abp.io/get-started) by easily selecting all options from the page.

Use the `new` command of the ABP CLI to create a new project:

````shell

abp new Acme.BookStore -t app-nolayers{{if UI == "NG"}} -u angular{{else if UI == "Blazor"}} -u blazor{{else if UI == "BlazorServer"}} -u blazor-server{{end}}{{if DB == "Mongo"}} -d mongodb{{end}}

````

*\*You can use different level of namespaces; e.g. BookStore, Acme.BookStore or Acme.Retail.BookStore.\**

> [ABP CLI document](./CLI.md) covers all of the available commands and options.

## The Solution Structure

The solution structure is based on the [Single-Layer Startup Template](Startup-Templates/Application-Single-Layer.md) where everything is in one project instead of the [Domain Driven Design](Domain-Driven-Design.md). You can check its [documentation](Startup-Templates/Application-Single-Layer.md) for more details.

{{ if DB == "Mongo" }}

## MongoDB Transactions

The [startup template](Startup-templates/Index.md) **\*\*disables\*\*** transactions in the `.MongoDB` project by default. If your MongoDB server supports transactions, you can enable it in the *\*YourProjectModule\** class's `ConfigureMongoDB` method:

```csharp

Configure<AbpUnitOfWorkDefaultOptions>(options =>

{

options.TransactionBehavior = UnitOfWorkTransactionBehavior.Enabled; //or UnitOfWorkTransactionBehavior.Auto

});

```

> Or you can delete that code since `Auto` is already the default behavior.

{{ end }}

## Next Step

\* [Running the solution](Getting-Started-Running-Solution-Single-Layer.md)

### 3: Running the Solution

# Getting Started

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

> This document assumes that you prefer to use **\*\*{{ UI\_Value }}\*\*** as the UI framework and **\*\*{{ DB\_Value }}\*\*** as the database provider. For other options, please change the preference on top of this document.

## Create the Database

### Connection String

Check the **\*\*connection string\*\*** in the `appsettings.json` file under the `YourProject` project.

{{ if DB == "EF" }}

````json

"ConnectionStrings": {

"Default": "Server=(LocalDb)\MSSQLLocalDB;Database=BookStore;Trusted\_Connection=True"

}

````

> **\*\*About the Connection Strings and Database Management Systems\*\***

>

> The solution is configured to use **\*\*Entity Framework Core\*\*** with **\*\*MS SQL Server\*\*** by default. However, if you've selected another DBMS using the `-dbms` parameter on the ABP CLI `new` command (like `-dbms MySQL`), the connection string might be different for you.

>

> EF Core supports [various](https://docs.microsoft.com/en-us/ef/core/providers/) database providers and you can use any supported DBMS. See [the Entity Framework integration document](Entity-Framework-Core.md) to learn how to [switch to another DBMS](Entity-Framework-Core-Other-DBMS.md) if you need later.

{{ else if DB == "Mongo" }}

````json

"ConnectionStrings": {

"Default": "mongodb://localhost:27017/BookStore"

}

````

The solution is configured to use **\*\*MongoDB\*\*** in your local computer, so you need to have a MongoDB server instance up and running or change the connection string to another MongoDB server.

{{ end }}

### Seed Initial Data

Before running the application, you need to create the database and seed the initial data. To do that, you can run the following command in the directory of your project (in the same folder of the `.csproj` file):

```bash

dotnet run --migrate-database

```

## Run the Application

{{if UI=="MVC" || UI=="BlazorServer"}}

Running the application is pretty straight-forward, you can run the application with any IDE that supports .NET or by running the `dotnet run` CLI command in the directory of your project:

{{else if UI=="Blazor"}}

Running the application is pretty straight-forward, you just need to run the `TodoApp.Host` application with any IDE that supports .NET or by running the `dotnet run` CLI command in the directory of your project.

> **\*\*Note:\*\*** The `host` application hosts and serves the `blazor` application. Therefore, you should run the `host` application only.

After the application runs, open the application in your default browser.

{{else if UI=="NG"}}

The solution has two main applications:

\* `TodoApp` (in the .NET solution) hosts the server-side HTTP API, so the Angular application can consume it. (server-side application)

\* `angular` folder contains the Angular application. (client-side application)

Firstly, run the `TodoApp` project in your favorite IDE (or run the `dotnet run` CLI command on your project directory) to see the server-side HTTP API on [Swagger UI](https://swagger.io/tools/swagger-ui/).

![swagger-ui](images/swagger-ui.png)

You can explore and test your HTTP API with this UI. If it works, then we can run the Angular client application.

You can run the application using the following (or `yarn start`) command:

````bash

npm start

````

This command takes time, but eventually runs and opens the application in your default browser.

{{end}}

After running the project, the index page should be seen as below:

![single-layer-index-page](images/single-layer-index-page.png)

Enter **\*\*admin\*\*** as the username and **\*\*1q2w3E\*\***\* as the password to login to the application. The application is up and running. You can start developing your application based on this startup template.

![bookstore-login-2](images/bookstore-login-2.png)

## Console Application

# Console Application Startup Template

This template is used to create a minimalist console application project.

## How to Start With?

First, install the [ABP CLI](../CLI.md) if you haven't installed before:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then use the `abp new` command in an empty folder to create a new solution:

````bash

abp new Acme.MyConsoleApp -t console

````

`Acme.MyConsoleApp` is the solution name, like *\*YourCompany.YourProduct\**. You can use single level, two-levels or three-levels naming.

## Solution Structure

After you use the above command to create a solution, you will have a solution like shown below:

![basic-console-application-solution](../images/basic-console-application-solution.png)

\* `HelloWorldService` is a sample service that implements the `ITransientDependency` interface to register this service to the [dependency injection](../Dependency-Injection.md) system.

## WPF Application

# WPF Application Startup Template

This template is used to create a minimalist WPF application project.

## How to Start With?

First, install the [ABP CLI](../CLI.md) if you haven't installed before:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then use the `abp new` command in an empty folder to create a new solution:

````bash

abp new Acme.MyWpfApp -t wpf

````

`Acme.MyWpfApp` is the solution name, like *\*YourCompany.YourProduct\**. You can use single level, two-levels or three-levels naming.

## Solution Structure

After you use the above command to create a solution, you will have a solution like shown below:

![basic-wpf-application-solution](../images/basic-wpf-application-solution.png)

\* `HelloWorldService` is a sample service that implements the `ITransientDependency` interface to register this service to the [dependency injection](../Dependency-Injection.md) system.

## MAUI

# MAUI Application Startup Template

This template is used to create a minimalist MAUI application project.

## How to Start With?

First, install the [ABP CLI](../CLI.md) if you haven't installed before:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then use the `abp new` command in an empty folder to create a new solution:

````bash

abp new Acme.MyMauiApp -t maui

````

`Acme.MyMauiApp` is the solution name, like *\*YourCompany.YourProduct\**. You can use single level, two-levels or three-levels naming.

## Solution Structure

After you use the above command to create a solution, you will have a solution like shown below:

![basic-maui-application-solution](../images/basic-maui-application-solution.png)

\* `HelloWorldService` is a sample service that implements the `ITransientDependency` interface to register this service to the [dependency injection](../Dependency-Injection.md) system.

## Empty Web Project

# Getting Started with an Empty ASP.NET Core MVC / Razor Pages Application

This tutorial explains how to start ABP from scratch with minimal dependencies. You generally want to start with the **\*\*[**startup template**](Getting-Started-AspNetCore-MVC-Template.md)\*\***.

## Create a New Project

1. Create a new AspNet Core Web Application with Visual Studio 2022 (17.0.0+):

![](images/create-new-aspnet-core-application-v2.png)

2. Configure your new project:

![](images/select-empty-web-application-v2.png)

3. Press the create button:

![create-aspnet-core-application](images/create-aspnet-core-application.png)

## Install Volo.Abp.AspNetCore.Mvc Package

You can use the [ABP CLI](CLI.md) to install the Volo.Abp.AspNetCore.Mvc package to your project. Execute the following command in the folder of the .csproj file that you want to install the package on:

````bash

abp add-package Volo.Abp.AspNetCore.Mvc

````

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.AspNetCore.Mvc).

## Create the First ABP Module

ABP is a modular framework and it requires a **\*\*startup (root) module\*\*** class derived from ``AbpModule``:

````C#

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.Hosting;

using Volo.Abp;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.Modularity;

namespace BasicAspNetCoreApplication

{

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

public class AppModule : AbpModule

{

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

var env = context.GetEnvironment();

// Configure the HTTP request pipeline.

if (env.IsDevelopment())

{

app.UseExceptionHandler("/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseConfiguredEndpoints();

}

}

}

````

``AppModule`` is a good name for the startup module for an application.

ABP packages define module classes and a module can depend on another. In the code above, the ``AppModule`` depends on the ``AbpAspNetCoreMvcModule`` (defined by the [Volo.Abp.AspNetCore.Mvc](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc) package). It's common to add a ``DependsOn`` attribute after installing a new ABP NuGet package.

Instead of the Startup class, we are configuring an ASP.NET Core pipeline in this module class.

## The Program Class

Next step is to modify the Program class to integrate to the ABP module system:

````C#

using BasicAspNetCoreApplication;

var builder = WebApplication.CreateBuilder(args);

await builder.AddApplicationAsync<AppModule>();

var app = builder.Build();

await app.InitializeApplicationAsync();

await app.RunAsync();

````

``builder.AddApplicationAsync<AppModule>();`` adds all services defined in all modules starting from the ``AppModule``.

``app.InitializeApplicationAsync()`` initializes and starts the application.

## Run the Application!

That's all! Run the application, it will just work as expected.

## Using Autofac as the Dependency Injection Framework

While ASP.NET Core's Dependency Injection (DI) system is fine for basic requirements, [Autofac](https://autofac.org/) provides advanced features like Property Injection and Method Interception which are required by ABP to perform advanced application framework features.

Replacing ASP.NET Core's DI system by Autofac and integrating to ABP is pretty easy.

1. Install [Volo.Abp.Autofac](https://www.nuget.org/packages/Volo.Abp.Autofac) package

````

Install-Package Volo.Abp.Autofac

````

2. Add the ``AbpAutofacModule`` Dependency

````C#

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

[DependsOn(typeof(AbpAutofacModule))] //Add dependency to ABP Autofac module

public class AppModule : AbpModule

{

...

}

````

3. Update `Program.cs` to use Autofac:

````C#

using BasicAspNetCoreApplication;

var builder = WebApplication.CreateBuilder(args);

builder.Host.UseAutofac(); //Add this line

await builder.AddApplicationAsync<AppModule>();

var app = builder.Build();

await app.InitializeApplicationAsync();

await app.RunAsync();

````

## Source Code

Get source code of the sample project created in this tutorial from [here](https://github.com/abpframework/abp-samples/tree/master/BasicAspNetCoreApplication).

# Tutorials

## Web Application Development

### 1: Creating the Server Side

# Web Application Development Tutorial - Part 1: Creating the Server Side

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the database provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts:

- **\*\*Part 1: Creating the server side (this part)\*\***

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

{{if UI == "MVC" && DB == "EF"}}

### Video Tutorial

This part is also recorded as a video tutorial and **\*\***<a href="https://www.youtube.com/watch?v=cJzyIFfAlp8&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=1" target="\_blank">**published on YouTube**</a>**\*\***.

{{end}}

## Creating the Solution

Before starting the development, create a new solution named `Acme.BookStore` and run it by following the [getting started tutorial](../Getting-Started.md).

## Create the Book Entity

**\*\*Domain layer\*\*** in the startup template is separated into two projects:

- `Acme.BookStore.Domain` contains your [entities](../Entities.md), [domain services](../Domain-Services.md) and other core domain objects.

- `Acme.BookStore.Domain.Shared` contains `constants`, `enums` or other domain related objects that can be shared with clients.

So, define your entities in the domain layer (`Acme.BookStore.Domain` project) of the solution.

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:

````csharp

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; }

}

````

\* ABP Framework has two fundamental base classes for entities: `AggregateRoot` and `Entity`. **\*\*Aggregate Root\*\*** is a [Domain Driven Design](../Domain-Driven-Design.md) concept which can be thought as a root entity that is directly queried and worked on (see the [entities document](../Entities.md) for more).

\* The `Book` entity inherits from the `AuditedAggregateRoot` which adds some base [auditing](../Audit-Logging.md) 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.

> This tutorial leaves the entity properties with **\*\*public get/set\*\*** for the sake of simplicity. See the [entities document](../Entities.md) if you want to learn more about DDD best practices.

### BookType Enum

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:

````csharp

namespace Acme.BookStore.Books;

public enum BookType

{

Undefined,

Adventure,

Biography,

Dystopia,

Fantastic,

Horror,

Science,

ScienceFiction,

Poetry

}

````

The final folder/file structure should be as shown below:

![bookstore-book-and-booktype](images/bookstore-book-and-booktype.png)

### Add the Book Entity to the DbContext

{{if DB == "EF"}}

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:

````csharp

public class BookStoreDbContext : AbpDbContext<BookStoreDbContext>

{

public DbSet<Book> Books { get; set; }

//...

}

````

{{end}}

{{if DB == "Mongo"}}

Add a `IMongoCollection<Book> Books` property to the `BookStoreMongoDbContext` inside the `Acme.BookStore.MongoDB` project:

```csharp

public class BookStoreMongoDbContext : AbpMongoDbContext

{

public IMongoCollection<Book> Books => Collection<Book>();

//...

}

```

{{end}}

{{if DB == "EF"}}

### Map the Book Entity to a Database Table

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

````csharp

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);

});

}

}

````

\* `BookStoreConsts` has constant values for the schema and table prefixes for your tables. You don't have to use it, but it's suggested to control the table prefixes in a single point.

\* The `ConfigureByConvention()` method gracefully configures/maps the inherited properties. Always use it for all your entities.

### Add Database Migration

The startup solution is configured to use [Entity Framework Core Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/). Since we've changed the database mapping configuration, we should create a new migration and apply changes to the database.

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

```bash

dotnet ef migrations add Created\_Book\_Entity

```

This will add a new migration class to the project:

![bookstore-efcore-migration](./images/bookstore-efcore-migration.png)

> 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.

{{end}}

### Add Sample Seed Data

> It's good to have some initial data in the database before running the application. This section introduces the [Data Seeding](../Data-Seeding.md) system of the ABP framework. You can skip this section if you don't want to create the data seeding, but it is suggested to follow along and learn this useful ABP Framework feature.

Create a class deriving from the `IDataSeedContributor` in the `\*.Domain` project by copying the following code:

```csharp

using 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

);

}

}

}

```

\* This code simply uses the `IRepository<Book, Guid>` (the default [repository](../Repositories.md)) to insert two books to the database in case there weren't any books in it.

### Update the Database

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

![bookstore-dbmigrator-on-solution](images/bookstore-dbmigrator-on-solution.png)

`.DbMigrator` is a console application that can be run to **\*\*migrate the database schema\*\*** and **\*\*seed the data\*\*** on **\*\*development\*\*** and **\*\*production\*\*** environments.

## Create the Application Service

The application layer is separated into two projects:

\* `Acme.BookStore.Application.Contracts` contains your [DTO](../Data-Transfer-Objects.md)s and [application service](../Application-Services.md) interfaces.

\* `Acme.BookStore.Application` contains the implementations of your application services.

In this section, you will create an application service to get, create, update and delete books using the `CrudAppService` base class of the ABP Framework.

### 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:

````csharp

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; }

}

````

\* **\*\*DTO\*\*** classes are used to **\*\*transfer data\*\*** between the *\*presentation layer\** and the *\*application layer\**. See the [Data Transfer Objects document](https://docs.abp.io/en/abp/latest/Data-Transfer-Objects) for more details.

\* The `BookDto` is used to transfer the book data to the presentation layer in order to show the book information on the UI.

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

It will be needed to map the `Book` entities to the `BookDto` objects while returning books to the presentation layer. [AutoMapper](https://automapper.org) 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:

````csharp

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore;

public class BookStoreApplicationAutoMapperProfile : Profile

{

public BookStoreApplicationAutoMapperProfile()

{

CreateMap<Book, BookDto>();

}

}

````

> See the [object to object mapping](../Object-To-Object-Mapping.md) document for details.

### CreateUpdateBookDto

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

````csharp

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.

\* It defines data annotation attributes (like `[Required]`) to define validations for the properties. `DTO`s are [automatically validated](https://docs.abp.io/en/abp/latest/Validation) by the ABP framework.

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:

````csharp

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore;

public class BookStoreApplicationAutoMapperProfile : Profile

{

public BookStoreApplicationAutoMapperProfile()

{

CreateMap<Book, BookDto>();

CreateMap<CreateUpdateBookDto, Book>();

}

}

````

### IBookAppService

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:

````csharp

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

{

}

````

\* 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).

### BookAppService

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:

````csharp

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](https://docs.abp.io/en/abp/latest/Repositories).

\* `BookAppService` uses `IObjectMapper` service ([see](../Object-To-Object-Mapping.md)) to map the `Book` objects to the `BookDto` objects and `CreateUpdateBookDto` objects to the `Book` objects. The Startup template uses the [AutoMapper](http://automapper.org/) library as the object mapping provider. We have defined the mappings before, so it will work as expected.

## Auto API Controllers

In a typical ASP.NET Core application, you create **\*\*API Controllers\*\*** to expose the application services as **\*\*HTTP API\*\*** endpoints. This allows browsers or 3rd-party clients to call them over HTTP.

ABP can [\*\*automagically\*\*](../API/Auto-API-Controllers.md) configure your application services as MVC API Controllers by convention.

### Swagger UI

The startup template is configured to run the [Swagger UI](https://swagger.io/tools/swagger-ui/) using the [Swashbuckle.AspNetCore](https://github.com/domaindrivendev/Swashbuckle.AspNetCore) library. Run the application ({{if UI=="MVC"}}`Acme.BookStore.Web`{{else}}`Acme.BookStore.HttpApi.Host`{{end}}) by pressing `CTRL+F5` and navigate to `https://localhost:<port>/swagger/` on your browser. Replace `<port>` with your own port number.

You will see some built-in service endpoints as well as the `Book` service and its REST-style endpoints:

![bookstore-swagger](./images/bookstore-swagger.png)

Swagger has a nice interface to test the APIs.

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

````json

{

"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"

}

]

}

````

That's pretty cool since we haven't written a single line of code to create the API controller, but now we have a fully working REST API!

## The Next Part

See the [next part](Part-2.md) of this tutorial.

### 2: The Book List Page

# Web Application Development Tutorial - Part 2: The Book List Page

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts:

- [Part 1: Creating the server side](Part-1.md)

- **\*\*Part 2: The book list page (this part)\*\***

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

{{if UI == "MVC" && DB == "EF"}}

### Video Tutorial

This part is also recorded as a video tutorial and **\*\***<a href="https://www.youtube.com/watch?v=UDNlLiPiBiw&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=2" target="\_blank">**published on YouTube**</a>**\*\***.

{{end}}

{{if UI == "MVC"}}

## Dynamic JavaScript Proxies

It's common to call the HTTP API endpoints via AJAX from the **\*\*JavaScript\*\*** side. You can use `$.ajax` or another tool to call the endpoints. However, ABP offers a better way.

ABP **\*\*dynamically\*\*** creates **\*\*[**JavaScript Proxies**](../UI/AspNetCore/Dynamic-JavaScript-Proxies.md)\*\*** for all the API endpoints. So, you can use any **\*\*endpoint\*\*** just like calling a **\*\*JavaScript function\*\***.

### Testing in the Developer Console

You can easily test the JavaScript proxies using your favorite browser's **\*\*Developer Console\*\***. Run the application, open your browser's **\*\*developer tools\*\*** (*\*shortcut is generally F12\**), switch to the **\*\*Console\*\*** tab, type the following code and press enter:

````js

acme.bookStore.books.book.getList({}).done(function (result) { console.log(result); });

````

\* `acme.bookStore.books` is the namespace of the `BookAppService` converted to [camelCase](https://en.wikipedia.org/wiki/Camel\_case).

\* `book` is the conventional name for the `BookAppService` (removed `AppService` postfix and converted to camelCase).

\* `getList` is the conventional name for the `GetListAsync` method defined in the `CrudAppService` base class (removed `Async` postfix and converted to camelCase).

\* The `{}` argument is used to send an empty object to the `GetListAsync` method which normally expects an object of type `PagedAndSortedResultRequestDto` that is used to send paging and sorting options to the server (all properties are optional with default values, so you can send an empty object).

\* The `getList` function returns a `promise`. You can pass a callback to the `then` (or `done`) function to get the result returned from the server.

Running this code produces the following output:

![bookstore-javascript-proxy-console](images/bookstore-javascript-proxy-console.png)

You can see the **\*\*book list\*\*** returned from the server. You can also check the **\*\*network\*\*** tab of the developer tools to see the client to server communication:

![bookstore-getlist-result-network](images/bookstore-getlist-result-network.png)

Let's **\*\*create a new book\*\*** using the `create` function:

````js

acme.bookStore.books.book.create({

name: 'Foundation',

type: 7,

publishDate: '1951-05-24',

price: 21.5

}).then(function (result) {

console.log('successfully created the book with id: ' + result.id);

});

````

> If you downloaded the source code of the tutorial and are following the steps from the sample, you should also pass the `authorId` parameter to the create method for **\*\*creating a new book\*\***.

You should see a message in the console that looks something like this:

````text

successfully created the book with id: 439b0ea8-923e-8e1e-5d97-39f2c7ac4246

````

Check the `Books` table in the database to see the new book row. You can try `get`, `update` and `delete` functions yourself.

We will use these dynamic proxy functions in the next sections to communicate with the server.

{{end}}

## Localization

Before starting the UI development, we first want to prepare the localization texts (you normally do this when needed while developing your application).

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

![bookstore-localization-files](images/bookstore-localization-files-v2.png)

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

````json

{

"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"

}

}

````

\* Localization key names are arbitrary. You can set any name. We prefer some conventions for specific text types;

\* Add `Menu:` prefix for menu items.

\* Use `Enum:<enum-type>.<enum-value>` or `<enum-type>.<enum-value>` naming convention to localize the enum members. When you do it like that, ABP can automatically localize the enums in some proper cases.

If a text is not defined in the localization file, it **\*\*falls back\*\*** to the localization key (as ASP.NET Core's standard behavior).

> ABP's localization system is built on the [ASP.NET Core's standard localization](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization) system and extends it in many ways. Check the [localization document](../Localization.md) for details.

{{if UI == "MVC"}}

## Create a Books Page

It's time to create something visible and usable! Instead of the classic MVC, we will use the [Razor Pages UI](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start) approach which is recommended by Microsoft.

Create a `Books` folder under the `Pages` folder of the `Acme.BookStore.Web` project. Add a new Razor Page by right clicking the Books folder then selecting **\*\*Add > Razor Page\*\*** menu item. Name it as `Index`:

![bookstore-add-index-page](images/bookstore-add-index-page-v2.png)

Open the `Index.cshtml` and change the whole content as shown below:

````html

@page

@using Acme.BookStore.Web.Pages.Books

@model IndexModel

<h2>Books</h2>

````

`Index.cshtml.cs` content should be like that:

```csharp

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Acme.BookStore.Web.Pages.Books;

public class IndexModel : PageModel

{

public void OnGet()

{

}

}

```

### Add Books Page to the Main Menu

Open the `BookStoreMenuContributor` class in the `Menus` folder and add the following code to the end of the `ConfigureMainMenuAsync` method:

````csharp

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 you can see that the new menu item has been added to the main menu:

![bookstore-menu-items](images/bookstore-new-menu-item-2.png)

When you click on the Books menu item under the Book Store parent, you will be redirected to the new empty Books Page.

### Book List

We will use the [Datatables.net](https://datatables.net/) jQuery library to show the book list. Datatables library completely works via AJAX, it is fast, popular and provides a good user experience.

> Datatables library is configured in the startup template, so you can directly use it in any page without including any style or script file for your page.

#### Index.cshtml

Change the `Pages/Books/Index.cshtml` as the following:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@model IndexModel

@inject IStringLocalizer<BookStoreResource> L

@section scripts

{

<abp-script src="/Pages/Books/Index.js" />

}

<abp-card>

<abp-card-header>

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

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="BooksTable"></abp-table>

</abp-card-body>

</abp-card>

````

\* `abp-script` [tag helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro) is used to add external **\*\*scripts\*\*** to the page. It has many additional features compared to the standard `script` tag. It handles **\*\*minification\*\*** and **\*\*versioning\*\***. Check the [bundling & minification document](../UI/AspNetCore/Bundling-Minification.md) for details.

\* `abp-card` is a tag helper for Twitter Bootstrap's [card component](https://getbootstrap.com/docs/4.5/components/card/). There are other useful tag helpers provided by the ABP Framework to easily use most of [bootstrap](https://getbootstrap.com/)'s components. You could use the regular HTML tags instead of these tag helpers, but using tag helpers reduces HTML code and prevents errors by the help of the IntelliSense and compiles time type checking. For further information, check the [tag helpers](../UI/AspNetCore/Tag-Helpers/Index.md) document.

#### Index.js

Create an `Index.js` file under the `Pages/Books` folder:

![bookstore-index-js-file](images/bookstore-index-js-file-v3.png)

The content of the file is shown below:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType.' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString(luxon.DateTime.DATETIME\_SHORT);

}

}

]

})

);

});

````

\* `abp.localization.getResource` gets a function that is used to localize text using the same JSON file defined on the server side. In this way, you can share the localization values with the client side.

\* `abp.libs.datatables.normalizeConfiguration` is a helper function defined by the ABP Framework. There's no requirement to use it, but it simplifies the [Datatables](https://datatables.net/) configuration by providing conventional default values for missing options.

\* `abp.libs.datatables.createAjax` is another helper function to adapt the ABP's dynamic JavaScript API proxies to the [Datatable](https://datatables.net/)'s expected parameter format

\* `acme.bookStore.books.book.getList` is the dynamic JavaScript proxy function introduced before.

\* [luxon](https://moment.github.io/luxon/) library is also a standard library that is pre-configured in the solution, so you can use to perform date/time operations easily.

> See [Datatables documentation](https://datatables.net/manual/) for all configuration options.

## Run the Final Application

You can run the application! The final UI of this part is shown below:

![Book list](images/bookstore-book-list-4.png)

This is a fully working, server side paged, sorted and localized table of books.

{{else if UI == "NG"}}

## Install NPM packages

> Notice: This tutorial is based on the ABP Framework v3.1.0+ If your project version is older, then please upgrade your solution. Check the [migration guide](../UI/Angular/Migration-Guide-v3.md) if you are upgrading an existing project with v2.x.

If you haven't done it before, open a new command line interface (terminal window) and go to your `angular` folder and then run the `yarn` command to install the NPM packages:

```bash

yarn

```

## Create a Books Page

It's time to create something visible and usable! There are some tools that we will use when developing the Angular frontend application:

- [Ng Bootstrap](https://ng-bootstrap.github.io/#/home) will be used as the UI component library.

- [Ngx-Datatable](https://swimlane.gitbook.io/ngx-datatable/) will be used as the datatable library.

Run the following command line to create a new module, named `BookModule` in the root folder of the angular application:

```bash

yarn ng generate module book --module app --routing --route books

```

This command should produce the following output:

````bash

> yarn ng generate module book --module app --routing --route books

yarn run v1.19.1

$ ng generate module book --module app --routing --route books

CREATE src/app/book/book-routing.module.ts (336 bytes)

CREATE src/app/book/book.module.ts (335 bytes)

CREATE src/app/book/book.component.html (19 bytes)

CREATE src/app/book/book.component.spec.ts (614 bytes)

CREATE src/app/book/book.component.ts (268 bytes)

CREATE src/app/book/book.component.scss (0 bytes)

UPDATE src/app/app-routing.module.ts (1289 bytes)

Done in 3.88s.

````

### BookModule

Open the `/src/app/book/book.module.ts` and replace the content as shown below:

````js

import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { BookRoutingModule } from './book-routing.module';

import { BookComponent } from './book.component';

@NgModule({

declarations: [BookComponent],

imports: [

BookRoutingModule,

SharedModule

]

})

export class BookModule { }

````

\* Added the `SharedModule`. `SharedModule` exports some common modules needed to create user interfaces.

\* `SharedModule` already exports the `CommonModule`, so we've removed the `CommonModule`.

### Routing

The generated code places the new route definition to the `src/app/app-routing.module.ts` file as shown below:

````js

const routes: Routes = [

// other route definitions...

{ path: 'books', loadChildren: () => import('./book/book.module').then(m => m.BookModule) },

];

````

Now, open the `src/app/route.provider.ts` file and replace the `configureRoutes` function declaration as shown below:

```js

function configureRoutes(routes: RoutesService) {

return () => {

routes.add([

{

path: '/',

name: '::Menu:Home',

iconClass: 'fas fa-home',

order: 1,

layout: eLayoutType.application,

},

{

path: '/book-store',

name: '::Menu:BookStore',

iconClass: 'fas fa-book',

order: 2,

layout: eLayoutType.application,

},

{

path: '/books',

name: '::Menu:Books',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

},

]);

};

}

```

`RoutesService` is a service provided by the ABP Framework to configure the main menu and the routes.

\* `path` is the URL of the route.

\* `name` is the localized menu item name (check the [localization document](../UI/Angular/Localization.md) for details).

\* `iconClass` is the icon of the menu item (you can use [Font Awesome](https://fontawesome.com/) icons by default).

\* `order` is the order of the menu item.

\* `layout` is the layout of the BooksModule's routes (there are three types of pre-defined layouts: `eLayoutType.application`, `eLayoutType.account` or `eLayoutType.empty`).

For more information, check the [RoutesService document](../UI/Angular/Modifying-the-Menu.md#via-routesservice).

### Service Proxy Generation

[ABP CLI](../CLI.md) provides a `generate-proxy` command that generates client proxies for your HTTP APIs to make your HTTP APIs easy to consume by the client side. Before running the `generate-proxy` command, your host must be up and running.

> **\*\*Warning\*\***: There is a problem with IIS Express; it doesn't allow connecting to the application from another process. If you are using Visual Studio, select the `Acme.BookStore.HttpApi.Host` instead of IIS Express in the run button drop-down list, as shown in the figure below:

![vs-run-without-iisexpress](images/vs-run-without-iisexpress.png)

Once the host application is running, execute the following command in the `angular` folder:

```bash

abp generate-proxy -t ng

```

This command will create the following files under the `/src/app/proxy/books` folder:

![Generated files](images/generated-proxies-3.png)

### BookComponent

Open the `/src/app/book/book.component.ts` file and replace the content as below:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto } from '@proxy/books';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

constructor(public readonly list: ListService, private bookService: BookService) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

}

```

\* We imported and injected the generated `BookService`.

\* We are using the [ListService](../UI/Angular/List-Service.md), a utility service from the ABP Framework which provides easy pagination, sorting and searching.

Open the `/src/app/book/book.component.html` and replace the content as shown below:

```html

<div class="card">

<div class="card-header">

<div class="row">

<div class="col col-md-6">

<h5 class="card-title">

{%{{{ '::Menu:Books' | abpLocalization }}}%}

</h5>

</div>

<div class="text-end col col-md-6"></div>

</div>

</div>

<div class="card-body">

<ngx-datatable [rows]="book.items" [count]="book.totalCount" [list]="list" default>

<ngx-datatable-column [name]="'::Name' | abpLocalization" prop="name"></ngx-datatable-column>

<ngx-datatable-column [name]="'::Type' | abpLocalization" prop="type">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ '::Enum:BookType.' + row.type | abpLocalization }}}%}

</ng-template>

</ngx-datatable-column>

<ngx-datatable-column [name]="'::PublishDate' | abpLocalization" prop="publishDate">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ row.publishDate | date }}}%}

</ng-template>

</ngx-datatable-column>

<ngx-datatable-column [name]="'::Price' | abpLocalization" prop="price">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ row.price | currency }}}%}

</ng-template>

</ngx-datatable-column>

</ngx-datatable>

</div>

</div>

```

Now you can see the final result on your browser:

![Book list final result](images/bookstore-book-list-angular.png)

{{else if UI == "Blazor" || UI == "BlazorServer"}}

## Create a Books Page

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.razor`:

![blazor-add-books-component](images/blazor-add-books-component.png)

Replace the contents of this component as shown below:

````html

@page "/books"

<h2>Books</h2>

@code {

}

````

### 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:

````csharp

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:

![blazor-menu-bookstore](images/bookstore-new-menu-item-2.png)

When you click on the Books menu item under the Book Store parent, you will be redirected to the new empty Books Page.

### Book List

We will use the [Blazorise library](https://blazorise.com/) as the UI component kit. It is a very powerful library that supports major HTML/CSS frameworks, including Bootstrap.

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:

````xml

@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>

````

> If you see some syntax errors, you can ignore them if your application is properly built and running. Visual Studio still has some bugs with Blazor.

\* 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.

While the code above is pretty easy to understand, you can check the Blazorise [Card](https://blazorise.com/docs/components/card/) and [DataGrid](https://blazorise.com/docs/extensions/datagrid/) documents to understand them better.

#### About the AbpCrudPageBase

We will continue benefitting from `AbpCrudPageBase` for the books page. You could just inject the `IBookAppService` and perform all the server side calls yourself (thanks to the [Dynamic C# HTTP API Client Proxy](../API/Dynamic-CSharp-API-Clients.md) system of the ABP Framework). We will do it manually for the authors page to demonstrate how to call the server side HTTP APIs in your Blazor applications.

## Run the Final Application

You can run the application! The final UI of this part is shown below:

![blazor-bookstore-book-list](images/blazor-bookstore-book-list-2.png)

This is a fully working, server side paged, sorted and localized table of books.

{{end # UI }}

## The Next Part

Check the [next part](Part-3.md) of this tutorial.

### 3: Creating, Updating and Deleting Books

# Web Application Development Tutorial - Part 3: Creating, Updating and Deleting Books

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts:

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- **\*\*Part 3: Creating, updating and deleting books (this part)\*\***

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

{{if UI == "MVC" && DB == "EF"}}

### Video Tutorial

This part is also recorded as a video tutorial and **\*\***<a href="https://www.youtube.com/watch?v=TLShZO8u2VE&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=3" target="\_blank">**published on YouTube**</a>**\*\***.

{{end}}

{{if UI == "MVC"}}

## Creating a New Book

In this section, you will learn how to create a new modal dialog form to create a new book. The modal dialog will look like the image below:

![bookstore-create-dialog](./images/bookstore-create-dialog-3.png)

### Create the Modal Form

Create a new razor page named `CreateModal.cshtml` under the `Pages/Books` folder of the `Acme.BookStore.Web` project.

![bookstore-add-create-dialog](./images/bookstore-add-create-dialog-v2.png)

#### CreateModal.cshtml.cs

Open the `CreateModal.cshtml.cs` file (`CreateModalModel` class) and replace it with the following code:

````C#

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

namespace Acme.BookStore.Web.Pages.Books

{

public class CreateModalModel : BookStorePageModel

{

[BindProperty]

public CreateUpdateBookDto Book { get; set; }

private readonly IBookAppService \_bookAppService;

public CreateModalModel(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public void OnGet()

{

Book = new CreateUpdateBookDto();

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.CreateAsync(Book);

return NoContent();

}

}

}

````

\* This class is derived from the `BookStorePageModel` instead of the standard `PageModel`. `BookStorePageModel` indirectly inherits the `PageModel` and adds some common properties & methods that can be shared in your page model classes.

\* `[BindProperty]` attribute on the `Book` property binds post request data to this property.

\* This class simply injects the `IBookAppService` in the constructor and calls the `CreateAsync` method in the `OnPostAsync` handler.

\* It creates a new `CreateUpdateBookDto` object in the `OnGet` method. ASP.NET Core can work without creating a new instance like that. However, it doesn't create an instance for you and if your class has some default value assignments or code execution in the class constructor, they won't work. For this case, we set default values for some of the `CreateUpdateBookDto` properties.

#### CreateModal.cshtml

Open the `CreateModal.cshtml` file and paste the code below:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model CreateModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<abp-dynamic-form abp-model="Book" asp-page="/Books/CreateModal">

<abp-modal>

<abp-modal-header title="@L["NewBook"].Value"></abp-modal-header>

<abp-modal-body>

<abp-form-content />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</abp-dynamic-form>

````

\* This modal uses `abp-dynamic-form` [tag helper](../UI/AspNetCore/Tag-Helpers/Dynamic-Forms.md) to automatically create the form from the `CreateUpdateBookDto` model class.

\* `abp-model` attribute indicates the model object where it's the `Book` property in this case.

\* `abp-form-content` tag helper is a placeholder to render the form controls (it is optional and needed only if you have added some other content in the `abp-dynamic-form` tag, just like in this page).

> Tip: `Layout` should be `null` just as done in this example since we don't want to include all the layout for the modals when they are loaded via AJAX.

### Add the "New book" Button

Open the `Pages/Books/Index.cshtml` and set the content of `abp-card-header` tag as below:

````html

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Books"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-end">

<abp-button id="NewBookButton"

text="@L["NewBook"].Value"

icon="plus"

button-type="Primary"/>

</abp-column>

</abp-row>

</abp-card-header>

````

The final content of `Index.cshtml` is shown below:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@model IndexModel

@inject IStringLocalizer<BookStoreResource> L

@section scripts

{

<abp-script src="/Pages/Books/Index.js"/>

}

<abp-card>

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Books"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-end">

<abp-button id="NewBookButton"

text="@L["NewBook"].Value"

icon="plus"

button-type="Primary"/>

</abp-column>

</abp-row>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="BooksTable"></abp-table>

</abp-card-body>

</abp-card>

````

This adds a new button called **\*\*New book\*\*** to the **\*\*top-right\*\*** of the table:

![bookstore-new-book-button](./images/bookstore-new-book-button-3.png)

Open the `Pages/Books/Index.js` file and add the following code right after the `Datatable` configuration:

````js

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

createModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

````

\* `abp.ModalManager` is a helper class to manage modals on the client side. It internally uses Twitter Bootstrap's standard modal, but abstracts many details by providing a simple API.

\* `createModal.onResult(...)` used to refresh the data table after creating a new book.

\* `createModal.open();` is used to open the model to create a new book.

The final content of the `Index.js` file should be like this:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType.' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

dataFormat: "datetime"

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

dataFormat: "datetime"

}

]

})

);

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

createModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

Now, you can **\*\*run the application\*\*** and add some new books using the new modal form.

## Updating a Book

Create a new razor page, named `EditModal.cshtml` under the `Pages/Books` folder of the `Acme.BookStore.Web` project:

![bookstore-add-edit-dialog](./images/bookstore-add-edit-dialog.png)

### EditModal.cshtml.cs

Open the `EditModal.cshtml.cs` file (`EditModalModel` class) and replace it with the following code:

````csharp

using System;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

namespace Acme.BookStore.Web.Pages.Books;

public class EditModalModel : BookStorePageModel

{

[HiddenInput]

[BindProperty(SupportsGet = true)]

public Guid Id { get; set; }

[BindProperty]

public CreateUpdateBookDto Book { get; set; }

private readonly IBookAppService \_bookAppService;

public EditModalModel(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public async Task OnGetAsync()

{

var bookDto = await \_bookAppService.GetAsync(Id);

Book = ObjectMapper.Map<BookDto, CreateUpdateBookDto>(bookDto);

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.UpdateAsync(Id, Book);

return NoContent();

}

}

````

\* `[HiddenInput]` and `[BindProperty]` are standard ASP.NET Core MVC attributes. `SupportsGet` is used to be able to get the `Id` value from the query string parameter of the request.

\* In the `OnGetAsync` method, we get the `BookDto` from the `BookAppService` and this is being mapped to the DTO object `CreateUpdateBookDto`.

\* The `OnPostAsync` uses `BookAppService.UpdateAsync(...)` to update the entity.

### Mapping from BookDto to CreateUpdateBookDto

To be able to map the `BookDto` to `CreateUpdateBookDto`, configure a new mapping. To do this, open the `BookStoreWebAutoMapperProfile.cs` file in the `Acme.BookStore.Web` project and change it as shown below:

````csharp

using AutoMapper;

namespace Acme.BookStore.Web;

public class BookStoreWebAutoMapperProfile : Profile

{

public BookStoreWebAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

}

}

````

\* We have just added `CreateMap<BookDto, CreateUpdateBookDto>();` to define this mapping.

> Notice that we do the mapping definition in the web layer as a best practice since it is only needed in this layer.

### EditModal.cshtml

Replace `EditModal.cshtml` content with the following content:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model EditModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<abp-dynamic-form abp-model="Book" asp-page="/Books/EditModal">

<abp-modal>

<abp-modal-header title="@L["Update"].Value"></abp-modal-header>

<abp-modal-body>

<abp-input asp-for="Id" />

<abp-form-content />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</abp-dynamic-form>

````

This page is very similar to `CreateModal.cshtml`, except:

\* It includes an `abp-input` for the `Id` property to store the `Id` of the editing book (which is a hidden input).

\* It uses `Books/EditModal` as the post URL.

### Add "Actions" Dropdown to the Table

We will add a dropdown button to the table named *\*Actions\**.

Open the `Pages/Books/Index.js` file and replace the content as below:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

var editModal = new abp.ModalManager(abp.appPath + 'Books/EditModal');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Actions'),

rowAction: {

items:

[

{

text: l('Edit'),

action: function (data) {

editModal.open({ id: data.record.id });

}

}

]

}

},

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType.' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

dataFormat: "datetime"

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

dataFormat: "datetime"

}

]

})

);

createModal.onResult(function () {

dataTable.ajax.reload();

});

editModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

\* Added a new `ModalManager` named `editModal` to open the edit modal dialog.

\* Added a new column at the beginning of the `columnDefs` section. This column is used for the "*\*Actions\**" dropdown button.

\* The "*\*Edit\**" action simply calls `editModal.open()` to open the edit dialog.

\* The `editModal.onResult(...)` callback refreshes the data table when you close the edit modal.

You can run the application and edit any book by selecting the edit action on a book.

The final UI looks as below:

![bookstore-books-table-actions](./images/bookstore-edit-button-3.png)

> Notice that you don't see the "Actions" button in the figure below. Instead, you see an "Edit" button. ABP is smart enough to show a single simple button instead of a actions dropdown button when the dropdown has only a single item. After the next section, it will turn to a drop down button.

## Deleting a Book

Open the `Pages/Books/Index.js` file and add a new item to the `rowAction` `items`:

````js

{

text: l('Delete'),

confirmMessage: function (data) {

return l('BookDeletionConfirmationMessage', data.record.name);

},

action: function (data) {

acme.bookStore.books.book

.delete(data.record.id)

.then(function() {

abp.notify.info(l('SuccessfullyDeleted'));

dataTable.ajax.reload();

});

}

}

````

\* The `confirmMessage` option is used to ask a confirmation question before executing the `action`.

\* The `acme.bookStore.books.book.delete(...)` method makes an AJAX request to the server to delete a book.

\* `abp.notify.info()` shows a notification after the delete operation.

Since we've used two new localization texts (`BookDeletionConfirmationMessage` and `SuccessfullyDeleted`) you need to add these to the localization file (`en.json` under the `Localization/BookStore` folder of the `Acme.BookStore.Domain.Shared` project):

````json

"BookDeletionConfirmationMessage": "Are you sure to delete the book '{0}'?",

"SuccessfullyDeleted": "Successfully deleted!"

````

The final `Index.js` content is shown below:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

var editModal = new abp.ModalManager(abp.appPath + 'Books/EditModal');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Actions'),

rowAction: {

items:

[

{

text: l('Edit'),

action: function (data) {

editModal.open({ id: data.record.id });

}

},

{

text: l('Delete'),

confirmMessage: function (data) {

return l(

'BookDeletionConfirmationMessage',

data.record.name

);

},

action: function (data) {

acme.bookStore.books.book

.delete(data.record.id)

.then(function() {

abp.notify.info(

l('SuccessfullyDeleted')

);

dataTable.ajax.reload();

});

}

}

]

}

},

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType.' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

dataFormat: "datetime"

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

dataFormat: "datetime"

}

]

})

);

createModal.onResult(function () {

dataTable.ajax.reload();

});

editModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

You can run the application and try to delete a book.

{{end}}

{{if UI == "NG"}}

## Creating a New Book

In this section, you will learn how to create a new modal dialog form to create a new book.

### BookComponent

Open `/src/app/book/book.component.ts` and replace the content as below:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto } from '@proxy/books';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

isModalOpen = false; // add this line

constructor(public readonly list: ListService, private bookService: BookService) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

// add new method

createBook() {

this.isModalOpen = true;

}

}

```

\* We defined a property called `isModalOpen` and a method called `createBook`.

Open `/src/app/book/book.component.html` and make the following changes:

```html

<div class="card">

<div class="card-header">

<div class="row">

<div class="col col-md-6">

<h5 class="card-title">{%{{{ '::Menu:Books' | abpLocalization }}}%}</h5>

</div>

<div class="text-end col col-md-6">

<!-- Add the "new book" button here -->

<div class="text-lg-end pt-2">

<button id="create" class="btn btn-primary" type="button" (click)="createBook()">

<i class="fa fa-plus me-1"></i>

<span>{%{{{ "::NewBook" | abpLocalization }}}%}</span>

</button>

</div>

</div>

</div>

</div>

<div class="card-body">

<!-- ngx-datatable should be here! -->

</div>

</div>

<!-- Add the modal here -->

<abp-modal [(visible)]="isModalOpen">

<ng-template #abpHeader>

<h3>{%{{{ '::NewBook' | abpLocalization }}}%}</h3>

</ng-template>

<ng-template #abpBody> </ng-template>

<ng-template #abpFooter>

<button type="button" class="btn btn-secondary" abpClose>

{%{{{ '::Close' | abpLocalization }}}%}

</button>

</ng-template>

</abp-modal>

```

\* Added a `New book` button to the card header..

\* Added the `abp-modal` which renders a modal to allow user to create a new book. `abp-modal` is a pre-built component to show modals. While you could use another approach to show a modal, `abp-modal` provides additional benefits.

You can open your browser and click the **\*\*New book\*\*** button to see the new modal.

![Empty modal for new book](./images/bookstore-empty-new-book-modal-2.png)

### Create a Reactive Form

[Reactive forms](https://angular.io/guide/reactive-forms) provide a model-driven approach to handling form inputs whose values change over time.

Open `/src/app/book/book.component.ts` and replace the content as below:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions } from '@proxy/books'; // add bookTypeOptions

import { FormGroup, FormBuilder, Validators } from '@angular/forms'; // add this

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

form: FormGroup; // add this line

// add bookTypes as a list of BookType enum members

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder // inject FormBuilder

) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.buildForm(); // add this line

this.isModalOpen = true;

}

// add buildForm method

buildForm() {

this.form = this.fb.group({

name: ['', Validators.required],

type: [null, Validators.required],

publishDate: [null, Validators.required],

price: [null, Validators.required],

});

}

// add save method

save() {

if (this.form.invalid) {

return;

}

this.bookService.create(this.form.value).subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

```

\* Imported `FormGroup`, `FormBuilder` and `Validators` from `@angular/forms`.

\* Added a `form: FormGroup` property.

\* Added a `bookTypes` property as a list of `BookType` enum members. That will be used in form options.

\* Injected `FormBuilder` into the constructor. [FormBuilder](https://angular.io/api/forms/FormBuilder) provides convenient methods for generating form controls. It reduces the amount of boilerplate needed to build complex forms.

\* Added a `buildForm` method to the end of the file and executed the `buildForm()` in the `createBook` method.

\* Added a `save` method.

Open `/src/app/book/book.component.html` and replace `<ng-template #abpBody> </ng-template>` with the following code part:

```html

<ng-template #abpBody>

<form [formGroup]="form" (ngSubmit)="save()">

<div class="mt-2">

<label for="book-name">Name</label><span> \* </span>

<input type="text" id="book-name" class="form-control" formControlName="name" autofocus />

</div>

<div class="mt-2">

<label for="book-price">Price</label><span> \* </span>

<input type="number" id="book-price" class="form-control" formControlName="price" />

</div>

<div class="mt-2">

<label for="book-type">Type</label><span> \* </span>

<select class="form-control" id="book-type" formControlName="type">

<option [ngValue]="null">Select a book type</option>

<option [ngValue]="type.value" \*ngFor="let type of bookTypes"> {%{{{ '::Enum:BookType.' + type.value | abpLocalization }}}%}</option>

</select>

</div>

<div class="mt-2">

<label>Publish date</label><span> \* </span>

<input

#datepicker="ngbDatepicker"

class="form-control"

name="datepicker"

formControlName="publishDate"

ngbDatepicker

(click)="datepicker.toggle()"

/>

</div>

</form>

</ng-template>

```

Also replace `<ng-template #abpFooter> </ng-template>` with the following code part:

````html

<ng-template #abpFooter>

<button type="button" class="btn btn-secondary" abpClose>

{%{{{ '::Close' | abpLocalization }}}%}

</button>

<!--added save button-->

<button class="btn btn-primary" (click)="save()" [disabled]="form.invalid">

<i class="fa fa-check mr-1"></i>

{%{{{ '::Save' | abpLocalization }}}%}

</button>

</ng-template>

````

### Datepicker

We've used [NgBootstrap datepicker](https://ng-bootstrap.github.io/#/components/datepicker/overview) in this component. So, we need to arrange the dependencies related to this component.

Open `/src/app/book/book.module.ts` and replace the content as below:

```js

import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { BookRoutingModule } from './book-routing.module';

import { BookComponent } from './book.component';

import { NgbDatepickerModule } from '@ng-bootstrap/ng-bootstrap'; // add this line

@NgModule({

declarations: [BookComponent],

imports: [

BookRoutingModule,

SharedModule,

NgbDatepickerModule, // add this line

]

})

export class BookModule { }

```

\* We imported `NgbDatepickerModule` to be able to use the date picker.

Open `/src/app/book/book.component.ts` and replace the content as below:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions } from '@proxy/books';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

// added this line

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [

ListService,

{ provide: NgbDateAdapter, useClass: NgbDateNativeAdapter } // add this line

],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

form: FormGroup;

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder

) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.buildForm();

this.isModalOpen = true;

}

buildForm() {

this.form = this.fb.group({

name: ['', Validators.required],

type: [null, Validators.required],

publishDate: [null, Validators.required],

price: [null, Validators.required],

});

}

save() {

if (this.form.invalid) {

return;

}

this.bookService.create(this.form.value).subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

```

\* Imported ` NgbDateNativeAdapter` and `NgbDateAdapter`.

\* We added a new provider `NgbDateAdapter` that converts the Datepicker value to `Date` type. Check out the [datepicker adapters](https://ng-bootstrap.github.io/#/components/datepicker/overview) for more details.

Now, you can open your browser to see the changes:

![Save button to the modal](./images/bookstore-new-book-form-v3.png)

## Updating a Book

Open `/src/app/book/book.component.ts` and replace the content as shown below:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions } from '@proxy/books';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService, { provide: NgbDateAdapter, useClass: NgbDateNativeAdapter }],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

selectedBook = {} as BookDto; // declare selectedBook

form: FormGroup;

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder

) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.selectedBook = {} as BookDto; // reset the selected book

this.buildForm();

this.isModalOpen = true;

}

// Add editBook method

editBook(id: string) {

this.bookService.get(id).subscribe((book) => {

this.selectedBook = book;

this.buildForm();

this.isModalOpen = true;

});

}

buildForm() {

this.form = this.fb.group({

name: [this.selectedBook.name || '', Validators.required],

type: [this.selectedBook.type || null, Validators.required],

publishDate: [

this.selectedBook.publishDate ? new Date(this.selectedBook.publishDate) : null,

Validators.required,

],

price: [this.selectedBook.price || null, Validators.required],

});

}

// change the save method

save() {

if (this.form.invalid) {

return;

}

const request = this.selectedBook.id

? this.bookService.update(this.selectedBook.id, this.form.value)

: this.bookService.create(this.form.value);

request.subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

```

\* We declared a variable named `selectedBook` as `BookDto`.

\* We added an `editBook` method. This method fetches the book with the given `id` and sets it to `selectedBook` object.

\* We replaced the `buildForm` method so that it creates the form with the `selectedBook` data.

\* We replaced the `createBook` method so it sets `selectedBook` to an empty object.

\* We changed the `save` method to handle both of create and update operations.

### Add "Actions" Dropdown to the Table

Open `/src/app/book/book.component.html`  and add the following `ngx-datatable-column` definition as the first column in the `ngx-datatable`:

```html

<ngx-datatable-column

[name]="'::Actions' | abpLocalization"

[maxWidth]="150"

[sortable]="false"

>

<ng-template let-row="row" ngx-datatable-cell-template>

<div ngbDropdown container="body" class="d-inline-block">

<button

class="btn btn-primary btn-sm dropdown-toggle"

data-toggle="dropdown"

aria-haspopup="true"

ngbDropdownToggle

>

<i class="fa fa-cog me-1"></i>{%{{{ '::Actions' | abpLocalization }}}%}

</button>

<div ngbDropdownMenu>

<button ngbDropdownItem (click)="editBook(row.id)">

{%{{{ '::Edit' | abpLocalization }}}%}

</button>

</div>

</div>

</ng-template>

</ngx-datatable-column>

```

Added an "Actions" dropdown as the first column of the table that is shown below:

![Action buttons](./images/bookstore-actions-buttons-2.png)

Also, change the `ng-template #abpHeader` section as shown below:

```html

<ng-template #abpHeader>

<h3>{%{{{ (selectedBook.id ? '::Edit' : '::NewBook' ) | abpLocalization }}}%}</h3>

</ng-template>

```

This template will show the **\*\*Edit\*\*** text for edit record operation, **\*\*New Book\*\*** for new record operation in the title.

## Deleting a Book

Open the `/src/app/book/book.component.ts` file and inject the `ConfirmationService`.

Replace the constructor as below:

```js

// ...

// add new imports

import { ConfirmationService, Confirmation } from '@abp/ng.theme.shared';

//change the constructor

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder,

private confirmation: ConfirmationService // inject the ConfirmationService

) {}

// Add a delete method

delete(id: string) {

this.confirmation.warn('::AreYouSureToDelete', '::AreYouSure').subscribe((status) => {

if (status === Confirmation.Status.confirm) {

this.bookService.delete(id).subscribe(() => this.list.get());

}

});

}

```

\* We imported `ConfirmationService`.

\* We injected `ConfirmationService` to the constructor.

\* Added a `delete` method.

> Check out the [Confirmation Popup documentation](../UI/Angular/Confirmation-Service) for more about this service.

### Add a Delete Button

Open `/src/app/book/book.component.html` and modify the `ngbDropdownMenu` to add the delete button as shown below:

```html

<div ngbDropdownMenu>

<!-- add the Delete button -->

<button ngbDropdownItem (click)="delete(row.id)">

{%{{{ '::Delete' | abpLocalization }}}%}

</button>

</div>

```

The final actions dropdown UI looks like below:

![bookstore-final-actions-dropdown](./images/bookstore-final-actions-dropdown-2.png)

Clicking the "Delete" action calls the `delete` method which then shows a confirmation popup as shown below:

![bookstore-confirmation-popup](./images/bookstore-confirmation-popup-2.png)

{{end}}

{{if UI == "Blazor" || UI == "BlazorServer"}}

## 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.

### Add a "New Button" Button

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

````xml

<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>

````

This will change the card header by adding a "New book" button to the right side:

![blazor-add-book-button](./images/blazor-add-book-button-2.png)

Now, we can add a modal that will be opened when we click the button.

### Book Creation Modal

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

````xml

<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>

````

This code requires a service; Inject the `AbpBlazorMessageLocalizerHelper<T>` at the top of the file, just before the `@inherits...` line:

````csharp

@inject AbpBlazorMessageLocalizerHelper<BookStoreResource> LH

````

\* The form implements validation and the `AbpBlazorMessageLocalizerHelper` is used to simply localize the validation messages.

\* The `CreateModal` object, `CloseCreateModalAsync` and `CreateEntityAsync` methods are defined by the base class. Check out the [Blazorise documentation](https://blazorise.com/docs/) if you want to understand the `Modal` and the other components.

That's all. Run the application and try to add a new book:

![blazor-new-book-modal](./images/blazor-new-book-modal-2.png)

## Updating a Book

Editing a book is similar to creating a new book.

### Actions Dropdown

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

````xml

<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:

![blazor-edit-book-action](./images/blazor-edit-book-action-3.png)

### Edit Modal

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

````xml

<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](../Object-To-Object-Mapping.md) 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:

````csharp

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore.Blazor;

public class BookStoreBlazorAutoMapperProfile : Profile

{

public BookStoreBlazorAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

}

}

````

\* We've just added the `CreateMap<BookDto, CreateUpdateBookDto>();` line to define the mapping.

### Test the Editing Modal

You can now run the application and try to edit a book.

![blazor-edit-book-modal](./images/blazor-edit-book-modal-2.png)

> Tip: Try to leave the *\*Name\** field empty and submit the form to show the validation error message.

## Deleting a Book

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

````xml

<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.

The "Actions" button becomes a dropdown since it has two actions now:

![blazor-delete-book-action](./images/blazor-delete-book-action-2.png)

Run the application and try to delete a book.

## Full CRUD UI Code

Here's the complete code to create the book management CRUD page, that has been developed in the last two parts:

````xml

@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>

````

{{end}}

## The Next Part

Check out the [next part](Part-4.md) of this tutorial.

### 4: Integration Tests

# Web Application Development Tutorial - Part 4: Integration Tests

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- **\*\*Part 4: Integration tests (this part)\*\***

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

{{if UI == "MVC" && DB == "EF"}}

### Video Tutorial

This part is also recorded as a video tutorial and **\*\***<a href="https://www.youtube.com/watch?v=aidRB4YFDLM&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=4" target="\_blank">**published on YouTube**</a>**\*\***.

{{end}}

## Test Projects in the Solution

This part covers the **\*\*server side\*\*** tests. There are several test projects in the solution:

![bookstore-test-projects-v2](./images/bookstore-test-projects-mvc.png)

> Test projects slightly differs based on your UI and Database selection. For example, if you select MongoDB, then the `Acme.BookStore.EntityFrameworkCore.Tests` will be `Acme.BookStore.MongoDB.Tests`.

Each project is used to test the related project. Test projects use the following libraries for testing:

\* [Xunit](https://github.com/xunit/xunit) as the main test framework.

\* [Shoudly](https://github.com/shouldly/shouldly) as the assertion library.

\* [NSubstitute](http://nsubstitute.github.io/) as the mocking library.

{{if DB=="EF"}}

> The test projects are configured to use **\*\*SQLite in-memory\*\*** as the database. A separate database instance is created and seeded (with the [data seed system](../Data-Seeding.md)) to prepare a fresh database for every test.

{{else if DB=="Mongo"}}

> **\*\*[**Mongo2Go**](https://github.com/Mongo2Go/Mongo2Go)\*\*** library is used to mock the MongoDB database. A separate database instance is created and seeded (with the [data seed system](../Data-Seeding.md)) to prepare a fresh database for every test.

{{end}}

## Adding Test Data

If you had created a data seed contributor as described in the [first part](Part-1.md), the same data will be available in your tests. So, you can skip this section. If you haven't created the seed contributor, you can use the `BookStoreTestDataSeedContributor` to seed the same data to be used in the tests below.

## Testing the BookAppService

Add a new test class, named `BookAppService\_Tests` in the `Books` namespace (folder) of the `Acme.BookStore.Application.Tests` project:

````csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Shouldly;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Validation;

using Xunit;

namespace Acme.BookStore.Books;

{{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]

{{end}}

public class BookAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IBookAppService \_bookAppService;

public BookAppService\_Tests()

{

\_bookAppService = GetRequiredService<IBookAppService>();

}

[Fact]

public async Task Should\_Get\_List\_Of\_Books()

{

//Act

var result = await \_bookAppService.GetListAsync(

new PagedAndSortedResultRequestDto()

);

//Assert

result.TotalCount.ShouldBeGreaterThan(0);

result.Items.ShouldContain(b => b.Name == "1984");

}

}

````

\* `Should\_Get\_List\_Of\_Books` test simply uses `BookAppService.GetListAsync` method to get and check the list of books.

\* We can safely check the book "1984" by its name, because we know that this books is available in the database since we've added it in the seed data.

Add a new test method to the `BookAppService\_Tests` class that creates a new **\*\*valid\*\*** book:

````csharp

[Fact]

public async Task Should\_Create\_A\_Valid\_Book()

{

//Act

var result = await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "New test book 42",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

//Assert

result.Id.ShouldNotBe(Guid.Empty);

result.Name.ShouldBe("New test book 42");

}

````

Add a new test that tries to create an invalid book and fails:

````csharp

[Fact]

public async Task Should\_Not\_Create\_A\_Book\_Without\_Name()

{

var exception = await Assert.ThrowsAsync<AbpValidationException>(async () =>

{

await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

});

exception.ValidationErrors

.ShouldContain(err => err.MemberNames.Any(mem => mem == "Name"));

}

````

\* Since the `Name` is empty, ABP will throw an `AbpValidationException`.

The final test class should be as shown below:

````csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Shouldly;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Validation;

using Xunit;

namespace Acme.BookStore.Books;

{{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]

{{end}}

public class BookAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IBookAppService \_bookAppService;

public BookAppService\_Tests()

{

\_bookAppService = GetRequiredService<IBookAppService>();

}

[Fact]

public async Task Should\_Get\_List\_Of\_Books()

{

//Act

var result = await \_bookAppService.GetListAsync(

new PagedAndSortedResultRequestDto()

);

//Assert

result.TotalCount.ShouldBeGreaterThan(0);

result.Items.ShouldContain(b => b.Name == "1984");

}

[Fact]

public async Task Should\_Create\_A\_Valid\_Book()

{

//Act

var result = await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "New test book 42",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

//Assert

result.Id.ShouldNotBe(Guid.Empty);

result.Name.ShouldBe("New test book 42");

}

[Fact]

public async Task Should\_Not\_Create\_A\_Book\_Without\_Name()

{

var exception = await Assert.ThrowsAsync<AbpValidationException>(async () =>

{

await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

});

exception.ValidationErrors

.ShouldContain(err => err.MemberNames.Any(mem => mem == "Name"));

}

}

````

Open the **\*\*Test Explorer Window\*\*** (use Test -> Windows -> Test Explorer menu if it is not visible) and **\*\*Run All\*\*** tests:

![bookstore-appservice-tests](./images/bookstore-appservice-tests.png)

Congratulations, the **\*\*green icons\*\*** indicates that the tests have been successfully passed!

## The Next Part

See the [next part](Part-5.md) of this tutorial.

### 5: Authorization

# Web Application Development Tutorial - Part 5: Authorization

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- **\*\*Part 5: Authorization (this part)\*\***

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

{{if UI == "MVC" && DB == "EF"}}

### Video Tutorial

This part is also recorded as a video tutorial and **\*\***<a href="https://www.youtube.com/watch?v=1WsfMITN\_Jk&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=5" target="\_blank">**published on YouTube**</a>**\*\***.

{{end}}

## Permissions

ABP Framework provides an [authorization system](../Authorization.md) based on the ASP.NET Core's [authorization infrastructure](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction). 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.

### 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:

````csharp

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.

### Permission Definitions

You should define permissions before using them.

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

````csharp

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:

````json

"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"

````

> Localization key names are arbitrary and there is no forcing rule. But we prefer the convention used above.

### Permission Management UI

Once you define the permissions, you can see them on the **\*\*permission management modal\*\***.

Go to the *\*Administration -> Identity -> Roles\** page, select *\*Permissions\** action for the admin role to open the permission management modal:

![bookstore-permissions-ui](images/bookstore-permissions-ui-2.png)

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.

## Authorization

Now, you can use the permissions to authorize the book management.

### Application Layer & HTTP API

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

````csharp

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](../API/Auto-API-Controllers.md)).

> You will see the declarative authorization, using the `[Authorize(...)]` attribute, later while developing the author management functionality.

{{if UI == "MVC"}}

### Razor Page

While securing the HTTP API & the application service prevents unauthorized users to use the services, they can still navigate to the book management page. While they will get authorization exception when the page makes the first AJAX call to the server, we should also authorize the page for a better user experience and security.

Open the `BookStoreWebModule` and add the following code block inside the `ConfigureServices` method:

````csharp

Configure<RazorPagesOptions>(options =>

{

options.Conventions.AuthorizePage("/Books/Index", BookStorePermissions.Books.Default);

options.Conventions.AuthorizePage("/Books/CreateModal", BookStorePermissions.Books.Create);

options.Conventions.AuthorizePage("/Books/EditModal", BookStorePermissions.Books.Edit);

});

````

Now, unauthorized users are redirected to the **\*\*login page\*\***.

#### Hide the New Book Button

The book management page has a *\*New Book\** button that should be invisible if the current user has no *\*Book Creation\** permission.

![bookstore-new-book-button-small](images/bookstore-new-book-button-small-2.png)

Open the `Pages/Books/Index.cshtml` file and change the content as shown below:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Permissions

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.AspNetCore.Authorization

@using Microsoft.Extensions.Localization

@model IndexModel

@inject IStringLocalizer<BookStoreResource> L

@inject IAuthorizationService AuthorizationService

@section scripts

{

<abp-script src="/Pages/Books/Index.js"/>

}

<abp-card>

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Books"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-end">

@if (await AuthorizationService.IsGrantedAsync(BookStorePermissions.Books.Create))

{

<abp-button id="NewBookButton"

text="@L["NewBook"].Value"

icon="plus"

button-type="Primary"/>

}

</abp-column>

</abp-row>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="BooksTable"></abp-table>

</abp-card-body>

</abp-card>

````

\* Added `@inject IAuthorizationService AuthorizationService` to access to the authorization service.

\* Used `@if (await AuthorizationService.IsGrantedAsync(BookStorePermissions.Books.Create))` to check the book creation permission to conditionally render the *\*New Book\** button.

### JavaScript Side

Books table in the book management page has an actions button for each row. The actions button includes *\*Edit\** and *\*Delete\** actions:

![bookstore-edit-delete-actions](images/bookstore-edit-delete-actions-2.png)

We should hide an action if the current user has not granted for the related permission. Datatables row actions has a `visible` option that can be set to `false` to hide the action item.

Open the `Pages/Books/Index.js` inside the `Acme.BookStore.Web` project and add a `visible` option to the `Edit` action as shown below:

````js

{

text: l('Edit'),

visible: abp.auth.isGranted('BookStore.Books.Edit'), //CHECK for the PERMISSION

action: function (data) {

editModal.open({ id: data.record.id });

}

}

````

Do same for the `Delete` action:

````js

visible: abp.auth.isGranted('BookStore.Books.Delete')

````

\* `abp.auth.isGranted(...)` is used to check a permission that is defined before.

\* `visible` could also be get a function that returns a `bool` if the value will be calculated later, based on some conditions.

### Menu Item

Even we have secured all the layers of the book management page, it is still visible on the main menu of the application. We should hide the menu item if the current user has no permission.

Open the `BookStoreMenuContributor` class, find the code block below:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

)

)

);

````

And replace this code block with the following:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

).RequirePermissions(BookStorePermissions.Books.Default) // Check the permission!

)

);

````

We've only added the `.RequirePermissions(BookStorePermissions.Books.Default)` extension method call for the inner menu item.

{{else if UI == "NG"}}

### Angular Guard Configuration

First step of the UI is to prevent unauthorized users to see the "Books" menu item and enter to the book management page.

Open the `/src/app/book/book-routing.module.ts` and replace with the following content:

````js

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { AuthGuard, PermissionGuard } from '@abp/ng.core';

import { BookComponent } from './book.component';

const routes: Routes = [

{ path: '', component: BookComponent, canActivate: [AuthGuard, PermissionGuard] },

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule],

})

export class BookRoutingModule {}

````

\* Imported `AuthGuard` and `PermissionGuard` from the `@abp/ng.core`.

\* Added `canActivate: [AuthGuard, PermissionGuard]` to the route definition.

Open the `/src/app/route.provider.ts` and add `requiredPolicy: 'BookStore.Books'` to the `/books` route. The `/books` route block should be following:

````js

{

path: '/books',

name: '::Menu:Books',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Books',

}

````

### Hide the New Book Button

The book management page has a *\*New Book\** button that should be invisible if the current user has no *\*Book Creation\** permission.

![bookstore-new-book-button-small](images/bookstore-new-book-button-small.png)

Open the `/src/app/book/book.component.html` file and replace the create button HTML content as shown below:

````html

<!-- Add the abpPermission directive -->

<button \*abpPermission="'BookStore.Books.Create'" id="create" class="btn btn-primary" type="button" (click)="createBook()">

<i class="fa fa-plus me-1"></i>

<span>{%{{{ '::NewBook' | abpLocalization }}}%}</span>

</button>

````

\* Just added `\*abpPermission="'BookStore.Books.Create'"` that hides the button if the current user has no permission.

### Hide the Edit and Delete Actions

Books table in the book management page has an actions button for each row. The actions button includes *\*Edit\** and *\*Delete\** actions:

![bookstore-edit-delete-actions](images/bookstore-edit-delete-actions-2.png)

We should hide an action if the current user has not granted for the related permission.

Open the `/src/app/book/book.component.html` file and replace the edit and delete buttons contents as shown below:

````html

<!-- Add the abpPermission directive -->

<button \*abpPermission="'BookStore.Books.Edit'" ngbDropdownItem (click)="editBook(row.id)">

{%{{{ '::Edit' | abpLocalization }}}%}

</button>

<!-- Add the abpPermission directive -->

<button \*abpPermission="'BookStore.Books.Delete'" ngbDropdownItem (click)="delete(row.id)">

{%{{{ '::Delete' | abpLocalization }}}%}

</button>

````

\* Added `\*abpPermission="'BookStore.Books.Edit'"` that hides the edit action if the current user has no editing permission.

\* Added `\*abpPermission="'BookStore.Books.Delete'"` that hides the delete action if the current user has no delete permission.

{{else if UI == "Blazor"}}

### Authorize the Razor Component

Open the `/Pages/Books.razor` file in the `Acme.BookStore.Blazor` project and add an `Authorize` attribute just after the `@page` directive and the following namespace imports (`@using` lines), as shown below:

````html

@page "/books"

@attribute [Authorize(BookStorePermissions.Books.Default)]

@using Acme.BookStore.Permissions

@using Microsoft.AspNetCore.Authorization

...

````

Adding this attribute prevents to enter this page if the current hasn't logged in or hasn't granted for the given permission. In case of attempt, the user is redirected to the login page.

### Show/Hide the Actions

The book management page has a *\*New Book\** button and *\*Edit\** and *\*Delete\** actions for each book. We should hide these buttons/actions if the current user has not granted for the related permissions.

The base `AbpCrudPageBase` class already has the necessary functionality for these kind of operations.

#### Set the Policy (Permission) Names

Add the following code block to the end of the `Books.razor` file:

````csharp

@code

{

public Books() // Constructor

{

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

}

````

The base `AbpCrudPageBase` class automatically checks these permissions on the related operations. It also defines the given properties for us if we need to check them manually:

\* `HasCreatePermission`: True, if the current user has permission to create the entity.

\* `HasUpdatePermission`: True, if the current user has permission to edit/update the entity.

\* `HasDeletePermission`: True, if the current user has permission to delete the entity.

> **\*\*Blazor Tip\*\***: While adding the C# code into a `@code` block is fine for small code parts, it is suggested to use the code behind approach to develop a more maintainable code base when the code block becomes longer. We will use this approach for the authors part.

#### Hide the New Book Button

Wrap the *\*New Book\** button by an `if` block as shown below:

````xml

@if (HasCreatePermission)

{

<Button Color="Color.Primary"

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

}

````

#### Hide the Edit/Delete Actions

`EntityAction` component defines `Visible` attribute (parameter) to conditionally show the action.

Update the `EntityActions` section as shown below:

````xml

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

<EntityAction TItem="BookDto"

Text="@L["Edit"]"

Visible=HasUpdatePermission

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

<EntityAction TItem="BookDto"

Text="@L["Delete"]"

Visible=HasDeletePermission

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

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

</EntityActions>

````

#### About the Permission Caching

You can run and test the permissions. Remove a book related permission from the admin role to see the related button/action disappears from the UI.

**\*\*ABP Framework caches the permissions\*\*** of the current user in the client side. So, when you change a permission for yourself, you need to manually **\*\*refresh the page\*\*** to take the effect. If you don't refresh and try to use the prohibited action you get an HTTP 403 (forbidden) response from the server.

> Changing a permission for a role or user immediately available on the server side. So, this cache system doesn't cause any security problem.

### Menu Item

Even we have secured all the layers of the book management page, it is still visible on the main menu of the application. We should hide the menu item if the current user has no permission.

Open the `BookStoreMenuContributor` class in the `Acme.BookStore.Blazor` project, find the code block below:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

)

)

);

````

And replace this code block with the following:

````csharp

var bookStoreMenu = new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

);

context.Menu.AddItem(bookStoreMenu);

//CHECK the PERMISSION

if (await context.IsGrantedAsync(BookStorePermissions.Books.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

));

}

````

You also need to add `async` keyword to the `ConfigureMenuAsync` method and re-arrange the return value. The final `ConfigureMainMenuAsync` method should be the following:

````csharp

private async Task ConfigureMainMenuAsync(MenuConfigurationContext context)

{

var l = context.GetLocalizer<BookStoreResource>();

context.Menu.Items.Insert(

0,

new ApplicationMenuItem(

"BookStore.Home",

l["Menu:Home"],

"/",

icon: "fas fa-home"

)

);

var bookStoreMenu = new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

);

context.Menu.AddItem(bookStoreMenu);

//CHECK the PERMISSION

if (await context.IsGrantedAsync(BookStorePermissions.Books.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

));

}

}

````

{{end}}

## The Next Part

See the [next part](Part-6.md) of this tutorial.

### 6: Authors: Domain layer

# Web Application Development Tutorial - Part 6: Authors: Domain Layer

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- **\*\*Part 6: Authors: Domain layer (this part)\*\***

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

## Introduction

In the previous parts, we've used the ABP infrastructure to easily build some services;

\* Used the [CrudAppService](../Application-Services.md) base class instead of manually developing an application service for standard create, read, update and delete operations.

\* Used [generic repositories](../Repositories.md) to completely automate the database layer.

For the "Authors" part;

\* We will **\*\*do some of the things manually\*\*** to show how you can do it in case of need.

\* We will implement some **\*\*Domain Driven Design (DDD) best practices\*\***.

> **\*\*The development will be done layer by layer to concentrate on an individual layer in one time. In a real project, you will develop your application feature by feature (vertical) as done in the previous parts. In this way, you will experience both approaches.\*\***

## The Author Entity

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

````csharp

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

);

}

}

````

\* Inherited from `FullAuditedAggregateRoot<Guid>` which makes the entity [soft delete](../Data-Filtering.md) (that means when you delete it, it is not deleted in the database, but just marked as deleted) with all the [auditing](../Entities.md) properties.

\* `private set` for the `Name` property restricts to set this property from out of this class. There are two ways of setting the name (in both cases, we validate the name):

\* In the constructor, while creating a new author.

\* Using the `ChangeName` method to update the name later.

\* The `constructor` and the `ChangeName` method is `internal` to force to use these methods only in the domain layer, using the `AuthorManager` that will be explained later.

\* `Check` class is an ABP Framework utility class to help you while checking method arguments (it throws `ArgumentException` on an invalid case).

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

````csharp

namespace Acme.BookStore.Authors;

public static class AuthorConsts

{

public const int MaxNameLength = 64;

}

````

Created this class inside the `Acme.BookStore.Domain.Shared` project since we will re-use it on the [Data Transfer Objects](../Data-Transfer-Objects.md) (DTOs) later.

## 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:

````csharp

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:

````csharp

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:

````csharp

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:

````json

"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.

## 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:

````csharp

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](../Repositories.md) 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.

We will implement this repository in the next part.

> Both of these methods might **\*\*seem unnecessary\*\*** since the standard repositories already provide generic querying methods and you can easily use them instead of defining such custom methods. You're right and do it like in a real application. However, for this **\*\*"learning" tutorial\*\***, it is useful to explain how to create custom repository methods when you really need it.

## Conclusion

This part covered the domain layer of the authors functionality of the book store application. The main files created/updated in this part was highlighted in the picture below:

![bookstore-author-domain-layer](images/bookstore-author-domain-layer.png)

## The Next Part

See the [next part](Part-7.md) of this tutorial.

### 7: Authors: Database Integration

# Web Application Development Tutorial - Part 7: Authors: Database Integration

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- **\*\*Part 7: Authors: Database Integration (this part)\*\***

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

## Introduction

This part explains how to configure the database integration for the `Author` entity introduced in the previous part.

{{if DB=="EF"}}

## DB Context

Open the `BookStoreDbContext` in the `Acme.BookStore.EntityFrameworkCore` project and add the following `DbSet` property:

````csharp

public DbSet<Author> Authors { get; set; }

````

Then locate to the `OnModelCreating` method in `BookStoreDbContext` class in the same project and add the following lines to the end of the method:

````csharp

builder.Entity<Author>(b =>

{

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

BookStoreConsts.DbSchema);

b.ConfigureByConvention();

b.Property(x => x.Name)

.IsRequired()

.HasMaxLength(AuthorConsts.MaxNameLength);

b.HasIndex(x => x.Name);

});

````

This is just like done for the `Book` entity before, so no need to explain again.

## Create a new Database Migration

The startup solution is configured to use [Entity Framework Core Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/). Since we've changed the database mapping configuration, we should create a new migration and apply changes to the database.

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

````bash

dotnet ef migrations add Added\_Authors

````

This will add a new migration class to the project:

![bookstore-efcore-migration-authors](./images/bookstore-efcore-migration-authors.png)

You can apply changes to the database using the following command, in the same command-line terminal:

````bash

dotnet ef database update

````

> If you are using Visual Studio, you may want to use the `Add-Migration Added\_Authors` 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.

{{else if DB=="Mongo"}}

## DB Context

Open the `BookStoreMongoDbContext` in the `MongoDb` folder of the `Acme.BookStore.MongoDB` project and add the following property to the class:

````csharp

public IMongoCollection<Author> Authors => Collection<Author>();

````

{{end}}

## Implementing the IAuthorRepository

{{if DB=="EF"}}

Create a new class, named `EfCoreAuthorRepository` inside the `Acme.BookStore.EntityFrameworkCore` project (in the `Authors` folder) and paste the following code:

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Threading.Tasks;

using Acme.BookStore.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using Volo.Abp.Domain.Repositories.EntityFrameworkCore;

using Volo.Abp.EntityFrameworkCore;

namespace Acme.BookStore.Authors;

public class EfCoreAuthorRepository

: EfCoreRepository<BookStoreDbContext, Author, Guid>,

IAuthorRepository

{

public EfCoreAuthorRepository(

IDbContextProvider<BookStoreDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task<Author> FindByNameAsync(string name)

{

var dbSet = await GetDbSetAsync();

return await dbSet.FirstOrDefaultAsync(author => author.Name == name);

}

public async Task<List<Author>> GetListAsync(

int skipCount,

int maxResultCount,

string sorting,

string filter = null)

{

var dbSet = await GetDbSetAsync();

return await dbSet

.WhereIf(

!filter.IsNullOrWhiteSpace(),

author => author.Name.Contains(filter)

)

.OrderBy(sorting)

.Skip(skipCount)

.Take(maxResultCount)

.ToListAsync();

}

}

````

\* Inherited from the `EfCoreRepository`, so it inherits the standard repository method implementations.

\* `WhereIf` is a shortcut extension method of the ABP Framework. It adds the `Where` condition only if the first condition meets (it filters by name, only if the filter was provided). You could do the same yourself, but these type of shortcut methods makes our life easier.

\* `sorting` can be a string like `Name`, `Name ASC` or `Name DESC`. It is possible by using the [System.Linq.Dynamic.Core](https://www.nuget.org/packages/System.Linq.Dynamic.Core) NuGet package.

> See the [EF Core Integration document](../Entity-Framework-Core.md) for more information on the EF Core based repositories.

{{else if DB=="Mongo"}}

Create a new class, named `MongoDbAuthorRepository` inside the `Acme.BookStore.MongoDB` project (in the `Authors` folder) and paste the following code:

```csharp

using System;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Collections.Generic;

using System.Threading.Tasks;

using Acme.BookStore.MongoDB;

using MongoDB.Driver;

using MongoDB.Driver.Linq;

using Volo.Abp.Domain.Repositories.MongoDB;

using Volo.Abp.MongoDB;

namespace Acme.BookStore.Authors;

public class MongoDbAuthorRepository

: MongoDbRepository<BookStoreMongoDbContext, Author, Guid>,

IAuthorRepository

{

public MongoDbAuthorRepository(

IMongoDbContextProvider<BookStoreMongoDbContext> dbContextProvider

) : base(dbContextProvider)

{

}

public async Task<Author> FindByNameAsync(string name)

{

var queryable = await GetMongoQueryableAsync();

return await queryable.FirstOrDefaultAsync(author => author.Name == name);

}

public async Task<List<Author>> GetListAsync(

int skipCount,

int maxResultCount,

string sorting,

string filter = null)

{

var queryable = await GetMongoQueryableAsync();

return await queryable

.WhereIf<Author, IMongoQueryable<Author>>(

!filter.IsNullOrWhiteSpace(),

author => author.Name.Contains(filter)

)

.OrderBy(sorting)

.As<IMongoQueryable<Author>>()

.Skip(skipCount)

.Take(maxResultCount)

.ToListAsync();

}

}

```

\* Inherited from the `MongoDbRepository`, so it inherits the standard repository method implementations.

\* `WhereIf` is a shortcut extension method of the ABP Framework. It adds the `Where` condition only if the first condition meets (it filters by name, only if the filter was provided). You could do the same yourself, but these type of shortcut methods makes our life easier.

\* `sorting` can be a string like `Name`, `Name ASC` or `Name DESC`. It is possible by using the [System.Linq.Dynamic.Core](https://www.nuget.org/packages/System.Linq.Dynamic.Core) NuGet package.

> See the [MongoDB Integration document](../MongoDB.md) for more information on the MongoDB based repositories.

{{end}}

## The Next Part

See the [next part](Part-8.md) of this tutorial.

### 8: Authors: Application Layer

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

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- **\*\*Part 8: Author: Application Layer (this part)\*\***

- [Part 9: Authors: User Interface](Part-9.md)

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

## Introduction

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

## IAuthorAppService

We will first create the [application service](../Application-Services.md) interface and the related [DTO](../Data-Transfer-Objects.md)s. Create a new interface, named `IAuthorAppService`, in the `Authors` namespace (folder) of the `Acme.BookStore.Application.Contracts` project:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

namespace Acme.BookStore.Authors;

public interface IAuthorAppService : IApplicationService

{

Task<AuthorDto> GetAsync(Guid id);

Task<PagedResultDto<AuthorDto>> GetListAsync(GetAuthorListDto input);

Task<AuthorDto> CreateAsync(CreateAuthorDto input);

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

````csharp

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>`.

### GetAuthorListDto

````csharp

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Authors;

public class GetAuthorListDto : PagedAndSortedResultRequestDto

{

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](../Data-Transfer-Objects.md) for all.

### CreateAuthorDto

````csharp

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](../Validation.md) for details.

### UpdateAuthorDto

````csharp

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:

````csharp

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;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Authors;

[Authorize(BookStorePermissions.Authors.Default)]

public class AuthorAppService : BookStoreAppService, IAuthorAppService

{

private readonly IAuthorRepository \_authorRepository;

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](../Authorization.md) 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

````csharp

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](../Object-To-Object-Mapping.md). This requires to configure the AutoMapper, which will be explained later.

### GetListAsync

````csharp

public async Task<PagedResultDto<AuthorDto>> GetListAsync(GetAuthorListDto input)

{

if (input.Sorting.IsNullOrWhiteSpace())

{

input.Sorting = nameof(Author.Name);

}

var authors = await \_authorRepository.GetListAsync(

input.SkipCount,

input.MaxResultCount,

input.Sorting,

input.Filter

);

var totalCount = input.Filter == null

? await \_authorRepository.CountAsync()

: await \_authorRepository.CountAsync(

author => author.Name.Contains(input.Filter));

return new PagedResultDto<AuthorDto>(

totalCount,

ObjectMapper.Map<List<Author>, List<AuthorDto>>(authors)

);

}

````

\* 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. If a filter is sent, then we are using it to filter entities while getting the count.

\* Finally, returning a paged result by mapping the list of `Author`s to a list of `AuthorDto`s.

### CreateAsync

````csharp

[Authorize(BookStorePermissions.Authors.Create)]

public async Task<AuthorDto> CreateAsync(CreateAuthorDto input)

{

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 `AuthorManager` (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

````csharp

[Authorize(BookStorePermissions.Authors.Edit)]

public async Task UpdateAsync(Guid id, UpdateAuthorDto input)

{

var author = await \_authorRepository.GetAsync(id);

if (author.Name != input.Name)

{

await \_authorManager.ChangeNameAsync(author, input.Name);

}

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.

{{if DB == "EF"}}

> **\*\*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.

{{end}}

### DeleteAsync

````csharp

[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:

````csharp

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";

}

// \*\*\* ADDED a NEW NESTED CLASS \*\*\*

public static class Authors

{

public const string Default = GroupName + ".Authors";

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

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

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

}

}

````

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

````csharp

var authorsPermission = bookStoreGroup.AddPermission(

BookStorePermissions.Authors.Default, L("Permission:Authors"));

authorsPermission.AddChild(

BookStorePermissions.Authors.Create, L("Permission:Authors.Create"));

authorsPermission.AddChild(

BookStorePermissions.Authors.Edit, L("Permission:Authors.Edit"));

authorsPermission.AddChild(

BookStorePermissions.Authors.Delete, L("Permission:Authors.Delete"));

````

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

````csharp

"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

`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:

````csharp

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:

````csharp

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> \_bookRepository;

private readonly IAuthorRepository \_authorRepository;

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 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

);

}

// ADDED SEED DATA FOR AUTHORS

if (await \_authorRepository.GetCountAsync() <= 0)

{

await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"George Orwell",

new DateTime(1903, 06, 25),

"Orwell produced literary criticism and poetry, fiction and polemical journalism; and is best known for the allegorical novella Animal Farm (1945) and the dystopian novel Nineteen Eighty-Four (1949)."

)

);

await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"Douglas Adams",

new DateTime(1952, 03, 11),

"Douglas Adams was an English author, screenwriter, essayist, humorist, satirist and dramatist. Adams was an advocate for environmentalism and conservation, a lover of fast cars, technological innovation and the Apple Macintosh, and a self-proclaimed 'radical atheist'."

)

);

}

}

}

````

{{if DB=="EF"}}

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

{{else if DB=="Mongo"}}

You can now run the `.DbMigrator` console application to **\*\*seed\*\*** the initial data.

{{end}}

## 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:

````csharp

using System;

using System.Threading.Tasks;

using Shouldly;

using Xunit;

namespace Acme.BookStore.Authors;

{{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]

{{end}}

public class AuthorAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IAuthorAppService \_authorAppService;

public AuthorAppService\_Tests()

{

\_authorAppService = GetRequiredService<IAuthorAppService>();

}

[Fact]

public async Task Should\_Get\_All\_Authors\_Without\_Any\_Filter()

{

var result = await \_authorAppService.GetListAsync(new GetAuthorListDto());

result.TotalCount.ShouldBeGreaterThanOrEqualTo(2);

result.Items.ShouldContain(author => author.Name == "George Orwell");

result.Items.ShouldContain(author => author.Name == "Douglas Adams");

}

[Fact]

public async Task Should\_Get\_Filtered\_Authors()

{

var result = await \_authorAppService.GetListAsync(

new GetAuthorListDto {Filter = "George"});

result.TotalCount.ShouldBeGreaterThanOrEqualTo(1);

result.Items.ShouldContain(author => author.Name == "George Orwell");

result.Items.ShouldNotContain(author => author.Name == "Douglas Adams");

}

[Fact]

public async Task Should\_Create\_A\_New\_Author()

{

var authorDto = await \_authorAppService.CreateAsync(

new CreateAuthorDto

{

Name = "Edward Bellamy",

BirthDate = new DateTime(1850, 05, 22),

ShortBio = "Edward Bellamy was an American author..."

}

);

authorDto.Id.ShouldNotBe(Guid.Empty);

authorDto.Name.ShouldBe("Edward Bellamy");

}

[Fact]

public async Task Should\_Not\_Allow\_To\_Create\_Duplicate\_Author()

{

await Assert.ThrowsAsync<AuthorAlreadyExistsException>(async () =>

{

await \_authorAppService.CreateAsync(

new CreateAuthorDto

{

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](Part-9.md) of this tutorial.

### 9: Authors: User Interface

# Web Application Development Tutorial - Part 9: Authors: User Interface

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- **\*\*Part 9: Authors: User Interface (this part)\*\***

- [Part 10: Book to Author Relation](Part-10.md)

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

## Introduction

This part explains how to create a CRUD page for the `Author` entity introduced in the previous parts.

{{if UI == "MVC"}}

## The Authors List Page

Create a new razor page, `Index.cshtml` under the `Pages/Authors` folder of the `Acme.BookStore.Web` project and change the content as given below.

### Index.cshtml

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Permissions

@using Acme.BookStore.Web.Pages.Authors

@using Microsoft.AspNetCore.Authorization

@using Microsoft.Extensions.Localization

@inject IStringLocalizer<BookStoreResource> L

@inject IAuthorizationService AuthorizationService

@model IndexModel

@section scripts

{

<abp-script src="/Pages/Authors/Index.js"/>

}

<abp-card>

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Authors"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-end">

@if (await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Create))

{

<abp-button id="NewAuthorButton"

text="@L["NewAuthor"].Value"

icon="plus"

button-type="Primary"/>

}

</abp-column>

</abp-row>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="AuthorsTable"></abp-table>

</abp-card-body>

</abp-card>

````

This is a simple page similar to the Books page we had created before. It imports a JavaScript file which will be introduced below.

### Index.cshtml.cs

````csharp

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Acme.BookStore.Web.Pages.Authors;

public class IndexModel : PageModel

{

public void OnGet()

{

}

}

````

### Index.js

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var createModal = new abp.ModalManager(abp.appPath + 'Authors/CreateModal');

var editModal = new abp.ModalManager(abp.appPath + 'Authors/EditModal');

var dataTable = $('#AuthorsTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.authors.author.getList),

columnDefs: [

{

title: l('Actions'),

rowAction: {

items:

[

{

text: l('Edit'),

visible:

abp.auth.isGranted('BookStore.Authors.Edit'),

action: function (data) {

editModal.open({ id: data.record.id });

}

},

{

text: l('Delete'),

visible:

abp.auth.isGranted('BookStore.Authors.Delete'),

confirmMessage: function (data) {

return l(

'AuthorDeletionConfirmationMessage',

data.record.name

);

},

action: function (data) {

acme.bookStore.authors.author

.delete(data.record.id)

.then(function() {

abp.notify.info(

l('SuccessfullyDeleted')

);

dataTable.ajax.reload();

});

}

}

]

}

},

{

title: l('Name'),

data: "name"

},

{

title: l('BirthDate'),

data: "birthDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

}

]

})

);

createModal.onResult(function () {

dataTable.ajax.reload();

});

editModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewAuthorButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

Briefly, this JavaScript page;

\* Creates a Data table with `Actions`, `Name` and `BirthDate` columns.

\* `Actions` column is used to add *\*Edit\** and *\*Delete\** actions.

\* `BirthDate` provides a `render` function to format the `DateTime` value using the [luxon](https://moment.github.io/luxon/) library.

\* Uses the `abp.ModalManager` to open *\*Create\** and *\*Edit\** modal forms.

This code is very similar to the Books page created before, so we will not explain it more.

### Localizations

This page uses some localization keys we need to declare. Open the `en.json` file under the `Localization/BookStore` folder of the `Acme.BookStore.Domain.Shared` project and add the following entries:

````json

"Menu:Authors": "Authors",

"Authors": "Authors",

"AuthorDeletionConfirmationMessage": "Are you sure to delete the author '{0}'?",

"BirthDate": "Birth date",

"NewAuthor": "New author"

````

Notice that we've added more keys. They will be used in the next sections.

### Add to the Main Menu

Open the `BookStoreMenuContributor.cs` in the `Menus` folder of the `Acme.BookStore.Web` project and add a new *\*Authors\** menu item under the *\*Book Store\** menu item. The following code (in the `ConfigureMainMenuAsync` method) shows the final code part:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

).RequirePermissions(BookStorePermissions.Books.Default)

).AddItem( // ADDED THE NEW "AUTHORS" MENU ITEM UNDER THE "BOOK STORE" MENU

new ApplicationMenuItem(

"BooksStore.Authors",

l["Menu:Authors"],

url: "/Authors"

).RequirePermissions(BookStorePermissions.Authors.Default)

)

);

````

### Run the Application

Run and login to the application. **\*\*You can not see the menu item since you don't have permission yet.\*\*** Go to the `Identity/Roles` page, click to the *\*Actions\** button and select the *\*Permissions\** action for the **\*\*admin role\*\***:

![bookstore-author-permissions](images/bookstore-author-permissions-3.png)

As you see, the admin role has no *\*Author Management\** permissions yet. Click to the checkboxes and save the modal to grant the necessary permissions. You will see the *\*Authors\** menu item under the *\*Book Store\** in the main menu, after **\*\*refreshing the page\*\***:

![bookstore-authors-page](images/bookstore-authors-page-3.png)

The page is fully working except *\*New author\** and *\*Actions/Edit\** since we haven't implemented them yet.

> **\*\*Tip\*\***: If you run the `.DbMigrator` console application after defining a new permission, it automatically grants these new permissions to the admin role and you don't need to manually grant the permissions yourself.

## Create Modal

Create a new razor page, `CreateModal.cshtml` under the `Pages/Authors` folder of the `Acme.BookStore.Web` project and change the content as given below.

### CreateModal.cshtml

```html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Authors

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model CreateModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<form asp-page="/Authors/CreateModal">

<abp-modal>

<abp-modal-header title="@L["NewAuthor"].Value"></abp-modal-header>

<abp-modal-body>

<abp-input asp-for="Author.Name" />

<abp-input asp-for="Author.BirthDate" />

<abp-input asp-for="Author.ShortBio" />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</form>

```

We had used [dynamic forms](../UI/AspNetCore/Tag-Helpers/Dynamic-Forms.md) of the ABP Framework for the books page before. We could use the same approach here, but we wanted to show how to do it manually. Actually, not so manually, because we've used `abp-input` tag helper in this case to simplify creating the form elements.

You can definitely use the standard Bootstrap HTML structure, but it requires to write a lot of code. `abp-input` automatically adds validation, localization and other standard elements based on the data type.

### CreateModal.cshtml.cs

```csharp

using System;

using System.ComponentModel.DataAnnotations;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Authors;

public class CreateModalModel : BookStorePageModel

{

[BindProperty]

public CreateAuthorViewModel Author { get; set; }

private readonly IAuthorAppService \_authorAppService;

public CreateModalModel(IAuthorAppService authorAppService)

{

\_authorAppService = authorAppService;

}

public void OnGet()

{

Author = new CreateAuthorViewModel();

}

public async Task<IActionResult> OnPostAsync()

{

var dto = ObjectMapper.Map<CreateAuthorViewModel, CreateAuthorDto>(Author);

await \_authorAppService.CreateAsync(dto);

return NoContent();

}

public class CreateAuthorViewModel

{

[Required]

[StringLength(AuthorConsts.MaxNameLength)]

public string Name { get; set; }

[Required]

[DataType(DataType.Date)]

public DateTime BirthDate { get; set; }

[TextArea]

public string ShortBio { get; set; }

}

}

```

This page model class simply injects and uses the `IAuthorAppService` to create a new author. The main difference between the book creation model class is that this one is declaring a new class, `CreateAuthorViewModel`, for the view model instead of re-using the `CreateAuthorDto`.

The main reason of this decision was to show you how to use a different model class inside the page. But there is one more benefit: We added two attributes to the class members, which were not present in the `CreateAuthorDto`:

\* Added `[DataType(DataType.Date)]` attribute to the `BirthDate` which shows a date picker on the UI for this property.

\* Added `[TextArea]` attribute to the `ShortBio` which shows a multi-line text area instead of a standard textbox.

In this way, you can specialize the view model class based on your UI requirements without touching to the DTO. As a result of this decision, we have used `ObjectMapper` to map `CreateAuthorViewModel` to `CreateAuthorDto`. To be able to do that, you need to add a new mapping code to the `BookStoreWebAutoMapperProfile` constructor:

````csharp

using Acme.BookStore.Authors; // ADDED NAMESPACE IMPORT

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore.Web;

public class BookStoreWebAutoMapperProfile : Profile

{

public BookStoreWebAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

// ADD a NEW MAPPING

CreateMap<Pages.Authors.CreateModalModel.CreateAuthorViewModel,

CreateAuthorDto>();

}

}

````

"New author" button will work as expected and open a new model when you run the application again:

![bookstore-new-author-modal](images/bookstore-new-author-modal-2.png)

## Edit Modal

Create a new razor page, `EditModal.cshtml` under the `Pages/Authors` folder of the `Acme.BookStore.Web` project and change the content as given below.

### EditModal.cshtml

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Authors

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model EditModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<form asp-page="/Authors/EditModal">

<abp-modal>

<abp-modal-header title="@L["Update"].Value"></abp-modal-header>

<abp-modal-body>

<abp-input asp-for="Author.Id" />

<abp-input asp-for="Author.Name" />

<abp-input asp-for="Author.BirthDate" />

<abp-input asp-for="Author.ShortBio" />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</form>

````

### EditModal.cshtml.cs

```csharp

using System;

using System.ComponentModel.DataAnnotations;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Authors;

public class EditModalModel : BookStorePageModel

{

[BindProperty]

public EditAuthorViewModel Author { get; set; }

private readonly IAuthorAppService \_authorAppService;

public EditModalModel(IAuthorAppService authorAppService)

{

\_authorAppService = authorAppService;

}

public async Task OnGetAsync(Guid id)

{

var authorDto = await \_authorAppService.GetAsync(id);

Author = ObjectMapper.Map<AuthorDto, EditAuthorViewModel>(authorDto);

}

public async Task<IActionResult> OnPostAsync()

{

await \_authorAppService.UpdateAsync(

Author.Id,

ObjectMapper.Map<EditAuthorViewModel, UpdateAuthorDto>(Author)

);

return NoContent();

}

public class EditAuthorViewModel

{

[HiddenInput]

public Guid Id { get; set; }

[Required]

[StringLength(AuthorConsts.MaxNameLength)]

public string Name { get; set; }

[Required]

[DataType(DataType.Date)]

public DateTime BirthDate { get; set; }

[TextArea]

public string ShortBio { get; set; }

}

}

```

This class is similar to the `CreateModal.cshtml.cs` while there are some main differences;

\* Uses the `IAuthorAppService.GetAsync(...)` method to get the editing author from the application layer.

\* `EditAuthorViewModel` has an additional `Id` property which is marked with the `[HiddenInput]` attribute that creates a hidden input for this property.

This class requires to add two object mapping declarations to the `BookStoreWebAutoMapperProfile` class:

```csharp

using Acme.BookStore.Authors;

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore.Web;

public class BookStoreWebAutoMapperProfile : Profile

{

public BookStoreWebAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

CreateMap<Pages.Authors.CreateModalModel.CreateAuthorViewModel,

CreateAuthorDto>();

// ADD THESE NEW MAPPINGS

CreateMap<AuthorDto, Pages.Authors.EditModalModel.EditAuthorViewModel>();

CreateMap<Pages.Authors.EditModalModel.EditAuthorViewModel,

UpdateAuthorDto>();

}

}

```

That's all! You can run the application and try to edit an author.

{{else if UI == "NG"}}

## The Author Management Page

Run the following command line to create a new module, named `AuthorModule` in the root folder of the angular application:

```bash

yarn ng generate module author --module app --routing --route authors

```

This command should produce the following output:

```bash

> yarn ng generate module author --module app --routing --route authors

yarn run v1.19.1

$ ng generate module author --module app --routing --route authors

CREATE src/app/author/author-routing.module.ts (344 bytes)

CREATE src/app/author/author.module.ts (349 bytes)

CREATE src/app/author/author.component.html (21 bytes)

CREATE src/app/author/author.component.spec.ts (628 bytes)

CREATE src/app/author/author.component.ts (276 bytes)

CREATE src/app/author/author.component.scss (0 bytes)

UPDATE src/app/app-routing.module.ts (1396 bytes)

Done in 2.22s.

```

### AuthorModule

Open the `/src/app/author/author.module.ts` and replace the content as shown below:

```js

import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { AuthorRoutingModule } from './author-routing.module';

import { AuthorComponent } from './author.component';

import { NgbDatepickerModule } from '@ng-bootstrap/ng-bootstrap';

@NgModule({

declarations: [AuthorComponent],

imports: [SharedModule, AuthorRoutingModule, NgbDatepickerModule],

})

export class AuthorModule {}

```

- Added the `SharedModule`. `SharedModule` exports some common modules needed to create user interfaces.

- `SharedModule` already exports the `CommonModule`, so we've removed the `CommonModule`.

- Added `NgbDatepickerModule` that will be used later on the author create and edit forms.

### Menu Definition

Open the `src/app/route.provider.ts` file and add the following menu definition:

````js

{

path: '/authors',

name: '::Menu:Authors',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Authors',

}

````

The final `configureRoutes` function declaration should be following:

```js

function configureRoutes(routes: RoutesService) {

return () => {

routes.add([

{

path: '/',

name: '::Menu:Home',

iconClass: 'fas fa-home',

order: 1,

layout: eLayoutType.application,

},

{

path: '/book-store',

name: '::Menu:BookStore',

iconClass: 'fas fa-book',

order: 2,

layout: eLayoutType.application,

},

{

path: '/books',

name: '::Menu:Books',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Books',

},

{

path: '/authors',

name: '::Menu:Authors',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Authors',

},

]);

};

}

```

### Service Proxy Generation

[ABP CLI](https://docs.abp.io/en/abp/latest/CLI) provides `generate-proxy` command that generates client proxies for your HTTP APIs to make easy to consume your HTTP APIs from the client side. Before running `generate-proxy` command, your host must be up and running.

Run the following command in the `angular` folder:

```bash

abp generate-proxy -t ng

```

This command generates the service proxy for the author service and the related model (DTO) classes:

![bookstore-angular-service-proxy-author](images/bookstore-angular-service-proxy-author-2.png)

### AuthorComponent

Open the `/src/app/author/author.component.ts` file and replace the content as below:

```js

import { Component, OnInit } from '@angular/core';

import { ListService, PagedResultDto } from '@abp/ng.core';

import { AuthorService, AuthorDto } from '@proxy/authors';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

import { ConfirmationService, Confirmation } from '@abp/ng.theme.shared';

@Component({

selector: 'app-author',

templateUrl: './author.component.html',

styleUrls: ['./author.component.scss'],

providers: [ListService, { provide: NgbDateAdapter, useClass: NgbDateNativeAdapter }],

})

export class AuthorComponent implements OnInit {

author = { items: [], totalCount: 0 } as PagedResultDto<AuthorDto>;

isModalOpen = false;

form: FormGroup;

selectedAuthor = {} as AuthorDto;

constructor(

public readonly list: ListService,

private authorService: AuthorService,

private fb: FormBuilder,

private confirmation: ConfirmationService

) {}

ngOnInit(): void {

const authorStreamCreator = (query) => this.authorService.getList(query);

this.list.hookToQuery(authorStreamCreator).subscribe((response) => {

this.author = response;

});

}

createAuthor() {

this.selectedAuthor = {} as AuthorDto;

this.buildForm();

this.isModalOpen = true;

}

editAuthor(id: string) {

this.authorService.get(id).subscribe((author) => {

this.selectedAuthor = author;

this.buildForm();

this.isModalOpen = true;

});

}

buildForm() {

this.form = this.fb.group({

name: [this.selectedAuthor.name || '', Validators.required],

birthDate: [

this.selectedAuthor.birthDate ? new Date(this.selectedAuthor.birthDate) : null,

Validators.required,

],

});

}

save() {

if (this.form.invalid) {

return;

}

if (this.selectedAuthor.id) {

this.authorService

.update(this.selectedAuthor.id, this.form.value)

.subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

} else {

this.authorService.create(this.form.value).subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

delete(id: string) {

this.confirmation.warn('::AreYouSureToDelete', '::AreYouSure')

.subscribe((status) => {

if (status === Confirmation.Status.confirm) {

this.authorService.delete(id).subscribe(() => this.list.get());

}

});

}

}

```

Open the `/src/app/author/author.component.html` and replace the content as below:

````html

<div class="card">

<div class="card-header">

<div class="row">

<div class="col col-md-6">

<h5 class="card-title">

{%{{{ '::Menu:Authors' | abpLocalization }}}%}

</h5>

</div>

<div class="text-end col col-md-6">

<div class="text-lg-end pt-2">

<button \*abpPermission="'BookStore.Authors.Create'" id="create" class="btn btn-primary" type="button" (click)="createAuthor()">

<i class="fa fa-plus me-1"></i>

<span>{%{{{ '::NewAuthor' | abpLocalization }}}%}</span>

</button>

</div>

</div>

</div>

</div>

<div class="card-body">

<ngx-datatable [rows]="author.items" [count]="author.totalCount" [list]="list" default>

<ngx-datatable-column

[name]="'::Actions' | abpLocalization"

[maxWidth]="150"

[sortable]="false"

>

<ng-template let-row="row" ngx-datatable-cell-template>

<div ngbDropdown container="body" class="d-inline-block">

<button

class="btn btn-primary btn-sm dropdown-toggle"

data-toggle="dropdown"

aria-haspopup="true"

ngbDropdownToggle

>

<i class="fa fa-cog me-1"></i>{%{{{ '::Actions' | abpLocalization }}}%}

</button>

<div ngbDropdownMenu>

<button \*abpPermission="'BookStore.Authors.Edit'" ngbDropdownItem (click)="editAuthor(row.id)">

{%{{{ '::Edit' | abpLocalization }}}%}

</button>

<button \*abpPermission="'BookStore.Authors.Delete'" ngbDropdownItem (click)="delete(row.id)">

{%{{{ '::Delete' | abpLocalization }}}%}

</button>

</div>

</div>

</ng-template>

</ngx-datatable-column>

<ngx-datatable-column [name]="'::Name' | abpLocalization" prop="name"></ngx-datatable-column>

<ngx-datatable-column [name]="'::BirthDate' | abpLocalization">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ row.birthDate | date }}}%}

</ng-template>

</ngx-datatable-column>

</ngx-datatable>

</div>

</div>

<abp-modal [(visible)]="isModalOpen">

<ng-template #abpHeader>

<h3>{%{{{ (selectedAuthor.id ? '::Edit' : '::NewAuthor') | abpLocalization }}}%}</h3>

</ng-template>

<ng-template #abpBody>

<form [formGroup]="form" (ngSubmit)="save()">

<div class="form-group">

<label for="author-name">Name</label><span> \* </span>

<input type="text" id="author-name" class="form-control" formControlName="name" autofocus />

</div>

<div class="mt-2">

<label>Birth date</label><span> \* </span>

<input

#datepicker="ngbDatepicker"

class="form-control"

name="datepicker"

formControlName="birthDate"

ngbDatepicker

(click)="datepicker.toggle()"

/>

</div>

</form>

</ng-template>

<ng-template #abpFooter>

<button type="button" class="btn btn-secondary" abpClose>

{%{{{ '::Close' | abpLocalization }}}%}

</button>

<button class="btn btn-primary" (click)="save()" [disabled]="form.invalid">

<i class="fa fa-check mr-1"></i>

{%{{{ '::Save' | abpLocalization }}}%}

</button>

</ng-template>

</abp-modal>

````

### Localizations

This page uses some localization keys we need to declare. Open the `en.json` file under the `Localization/BookStore` folder of the `Acme.BookStore.Domain.Shared` project and add the following entries:

````json

"Menu:Authors": "Authors",

"Authors": "Authors",

"AuthorDeletionConfirmationMessage": "Are you sure to delete the author '{0}'?",

"BirthDate": "Birth date",

"NewAuthor": "New author"

````

### Run the Application

Run and login to the application. **\*\*You can not see the menu item since you don't have permission yet.\*\*** Go to the `identity/roles` page, click to the *\*Actions\** button and select the *\*Permissions\** action for the **\*\*admin role\*\***:

![bookstore-author-permissions](images/bookstore-author-permissions-2.png)

As you see, the admin role has no *\*Author Management\** permissions yet. Click to the checkboxes and save the modal to grant the necessary permissions. You will see the *\*Authors\** menu item under the *\*Book Store\** in the main menu, after **\*\*refreshing the page\*\***:

![bookstore-authors-page](images/bookstore-angular-authors-page-2.png)

That's all! This is a fully working CRUD page, you can create, edit and delete authors.

> **\*\*Tip\*\***: If you run the `.DbMigrator` console application after defining a new permission, it automatically grants these new permissions to the admin role and you don't need to manually grant the permissions yourself.

{{end}}

{{if UI == "Blazor" || UI == "BlazorServer"}}

## The Author Management Page

### Authors Razor Component

Create a new Razor Component Page, `/Pages/Authors.razor`, in the `Acme.BookStore.Blazor` project with the following content:

````xml

@page "/authors"

@using Acme.BookStore.Authors

@using Acme.BookStore.Localization

@using Volo.Abp.AspNetCore.Components.Web

@inherits BookStoreComponentBase

@inject IAuthorAppService AuthorAppService

@inject AbpBlazorMessageLocalizerHelper<BookStoreResource> LH

<Card>

<CardHeader>

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

<Column ColumnSize="ColumnSize.IsAuto">

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

</Column>

<Column ColumnSize="ColumnSize.IsAuto">

@if (CanCreateAuthor)

{

<Button Color="Color.Primary"

Clicked="OpenCreateAuthorModal">

@L["NewAuthor"]

</Button>

}

</Column>

</Row>

</CardHeader>

<CardBody>

<DataGrid TItem="AuthorDto"

Data="AuthorList"

ReadData="OnDataGridReadAsync"

TotalItems="TotalCount"

ShowPager="true"

PageSize="PageSize">

<DataGridColumns>

<DataGridColumn Width="150px"

TItem="AuthorDto"

Field="@nameof(AuthorDto.Id)"

Sortable="false"

Caption="@L["Actions"]">

<DisplayTemplate>

<Dropdown>

<DropdownToggle Color="Color.Primary">

@L["Actions"]

</DropdownToggle>

<DropdownMenu>

@if (CanEditAuthor)

{

<DropdownItem Clicked="() => OpenEditAuthorModal(context)">

@L["Edit"]

</DropdownItem>

}

@if (CanDeleteAuthor)

{

<DropdownItem Clicked="() => DeleteAuthorAsync(context)">

@L["Delete"]

</DropdownItem>

}

</DropdownMenu>

</Dropdown>

</DisplayTemplate>

</DataGridColumn>

<DataGridColumn TItem="AuthorDto"

Field="@nameof(AuthorDto.Name)"

Caption="@L["Name"]"></DataGridColumn>

<DataGridColumn TItem="AuthorDto"

Field="@nameof(AuthorDto.BirthDate)"

Caption="@L["BirthDate"]">

<DisplayTemplate>

@context.BirthDate.ToShortDateString()

</DisplayTemplate>

</DataGridColumn>

</DataGridColumns>

</DataGrid>

</CardBody>

</Card>

<Modal @ref="CreateAuthorModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@L["NewAuthor"]</ModalTitle>

<CloseButton Clicked="CloseCreateAuthorModal" />

</ModalHeader>

<ModalBody>

<Validations @ref="@CreateValidationsRef" Model="@NewAuthor" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

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

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

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

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

<DateEdit TValue="DateTime" @bind-Date="@NewAuthor.BirthDate"/>

</Field>

<Validation MessageLocalizer="@LH.Localize">

<Field>

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

<MemoEdit Rows="5" @bind-Text="@NewAuthor.ShortBio">

<Feedback>

<ValidationError/>

</Feedback>

</MemoEdit>

</Field>

</Validation>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseCreateAuthorModal">

@L["Cancel"]

</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="CreateAuthorAsync">

@L["Save"]

</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

<Modal @ref="EditAuthorModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@EditingAuthor.Name</ModalTitle>

<CloseButton Clicked="CloseEditAuthorModal" />

</ModalHeader>

<ModalBody>

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

<Validation MessageLocalizer="@LH.Localize">

<Field>

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

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

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

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

<DateEdit TValue="DateTime" @bind-Date="@EditingAuthor.BirthDate"/>

</Field>

<Validation>

<Field>

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

<MemoEdit Rows="5" @bind-Text="@EditingAuthor.ShortBio">

<Feedback>

<ValidationError/>

</Feedback>

</MemoEdit>

</Field>

</Validation>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseEditAuthorModal">

@L["Cancel"]

</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="UpdateAuthorAsync">

@L["Save"]

</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

````

\* This code is similar to the `Books.razor`, except it doesn't inherit from the `AbpCrudPageBase`, but uses its own implementation.

\* Injects the `IAuthorAppService` to consume the server side HTTP APIs from the UI. We can directly inject application service interfaces and use just like regular method calls by the help of [Dynamic C# HTTP API Client Proxy System](../API/Dynamic-CSharp-API-Clients.md), which performs REST API calls for us. See the `Authors` class below to see the usage.

\* Injects the `IAuthorizationService` to check [permissions](../Authorization.md).

\* Injects the `IObjectMapper` for [object to object mapping](../Object-To-Object-Mapping.md).

Create a new code behind file, `Authors.razor.cs`, under the `Pages` folder, with the following content:

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Permissions;

using Blazorise;

using Blazorise.DataGrid;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Blazor.Pages;

public partial class Authors

{

private IReadOnlyList<AuthorDto> AuthorList { get; set; }

private int PageSize { get; } = LimitedResultRequestDto.DefaultMaxResultCount;

private int CurrentPage { get; set; }

private string CurrentSorting { get; set; }

private int TotalCount { get; set; }

private bool CanCreateAuthor { get; set; }

private bool CanEditAuthor { get; set; }

private bool CanDeleteAuthor { get; set; }

private CreateAuthorDto NewAuthor { get; set; }

private Guid EditingAuthorId { get; set; }

private UpdateAuthorDto EditingAuthor { get; set; }

private Modal CreateAuthorModal { get; set; }

private Modal EditAuthorModal { get; set; }

private Validations CreateValidationsRef;

private Validations EditValidationsRef;

public Authors()

{

NewAuthor = new CreateAuthorDto();

EditingAuthor = new UpdateAuthorDto();

}

protected override async Task OnInitializedAsync()

{

await SetPermissionsAsync();

await GetAuthorsAsync();

}

private async Task SetPermissionsAsync()

{

CanCreateAuthor = await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Create);

CanEditAuthor = await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Edit);

CanDeleteAuthor = await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Delete);

}

private async Task GetAuthorsAsync()

{

var result = await AuthorAppService.GetListAsync(

new GetAuthorListDto

{

MaxResultCount = PageSize,

SkipCount = CurrentPage \* PageSize,

Sorting = CurrentSorting

}

);

AuthorList = result.Items;

TotalCount = (int)result.TotalCount;

}

private async Task OnDataGridReadAsync(DataGridReadDataEventArgs<AuthorDto> e)

{

CurrentSorting = e.Columns

.Where(c => c.SortDirection != SortDirection.Default)

.Select(c => c.Field + (c.SortDirection == SortDirection.Descending ? " DESC" : ""))

.JoinAsString(",");

CurrentPage = e.Page - 1;

await GetAuthorsAsync();

await InvokeAsync(StateHasChanged);

}

private void OpenCreateAuthorModal()

{

CreateValidationsRef.ClearAll();

NewAuthor = new CreateAuthorDto();

CreateAuthorModal.Show();

}

private void CloseCreateAuthorModal()

{

CreateAuthorModal.Hide();

}

private void OpenEditAuthorModal(AuthorDto author)

{

EditValidationsRef.ClearAll();

EditingAuthorId = author.Id;

EditingAuthor = ObjectMapper.Map<AuthorDto, UpdateAuthorDto>(author);

EditAuthorModal.Show();

}

private async Task DeleteAuthorAsync(AuthorDto author)

{

var confirmMessage = L["AuthorDeletionConfirmationMessage", author.Name];

if (!await Message.Confirm(confirmMessage))

{

return;

}

await AuthorAppService.DeleteAsync(author.Id);

await GetAuthorsAsync();

}

private void CloseEditAuthorModal()

{

EditAuthorModal.Hide();

}

private async Task CreateAuthorAsync()

{

if (await CreateValidationsRef.ValidateAll())

{

await AuthorAppService.CreateAsync(NewAuthor);

await GetAuthorsAsync();

CreateAuthorModal.Hide();

}

}

private async Task UpdateAuthorAsync()

{

if (await EditValidationsRef.ValidateAll())

{

await AuthorAppService.UpdateAsync(EditingAuthorId, EditingAuthor);

await GetAuthorsAsync();

EditAuthorModal.Hide();

}

}

}

````

This class typically defines the properties and methods used by the `Authors.razor` page.

### Object Mapping

`Authors` class uses the `IObjectMapper` in the `OpenEditAuthorModal` method. So, we need to define this mapping.

Open the `BookStoreBlazorAutoMapperProfile.cs` in the `Acme.BookStore.Blazor` project and add the following mapping code in the constructor:

````csharp

CreateMap<AuthorDto, UpdateAuthorDto>();

````

You will need to declare a `using Acme.BookStore.Authors;` statement to the beginning of the file.

### Add to the Main Menu

Open the `BookStoreMenuContributor.cs` in the `Acme.BookStore.Blazor` project and add the following code to the end of the `ConfigureMainMenuAsync` method:

````csharp

if (await context.IsGrantedAsync(BookStorePermissions.Authors.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Authors",

l["Menu:Authors"],

url: "/authors"

));

}

````

### Localizations

We should complete the localizations we've used above. Open the `en.json` file under the `Localization/BookStore` folder of the `Acme.BookStore.Domain.Shared` project and add the following entries:

````json

"Menu:Authors": "Authors",

"Authors": "Authors",

"AuthorDeletionConfirmationMessage": "Are you sure to delete the author '{0}'?",

"BirthDate": "Birth date",

"NewAuthor": "New author"

````

### Run the Application

Run and login to the application. **\*\*If you don't see the Authors menu item under the Book Store menu, that means you don't have the permission yet.\*\*** Go to the `identity/roles` page, click to the *\*Actions\** button and select the *\*Permissions\** action for the **\*\*admin role\*\***:

![bookstore-author-permissions](images/bookstore-author-permissions-2.png)

As you see, the admin role has no *\*Author Management\** permissions yet. Click to the checkboxes and save the modal to grant the necessary permissions. You will see the *\*Authors\** menu item under the *\*Book Store\** in the main menu, after **\*\*refreshing the page\*\***:

![bookstore-authors-page](images/bookstore-authors-page-3.png)

That's all! This is a fully working CRUD page, you can create, edit and delete the authors.

> **\*\*Tip\*\***: If you run the `.DbMigrator` console application after defining a new permission, it automatically grants these new permissions to the admin role and you don't need to manually grant the permissions yourself.

{{end}}

## The Next Part

See the [next part](Part-10.md) of this tutorial.

### 10: Book to Author Relation

# Web Application Development Tutorial - Part 10: Book to Author Relation

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 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:

\* **\*\*{{DB\_Value}}\*\*** as the ORM provider.

\* **\*\*{{UI\_Value}}\*\*** as the UI Framework.

This tutorial is organized as the following parts;

- [Part 1: Creating the server side](Part-1.md)

- [Part 2: The book list page](Part-2.md)

- [Part 3: Creating, updating and deleting books](Part-3.md)

- [Part 4: Integration tests](Part-4.md)

- [Part 5: Authorization](Part-5.md)

- [Part 6: Authors: Domain layer](Part-6.md)

- [Part 7: Authors: Database Integration](Part-7.md)

- [Part 8: Authors: Application Layer](Part-8.md)

- [Part 9: Authors: User Interface](Part-9.md)

- **\*\*Part 10: Book to Author Relation (this part)\*\***

### 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](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI with EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI with MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> If you encounter the "filename too long" or "unzip" error on Windows, please see [this guide](../KB/Windows-Path-Too-Long-Fix.md).

## Introduction

We have created `Book` and `Author` functionalities for the book store application. However, currently there is no relation between these entities.

In this tutorial, we will establish a **\*\*1 to N\*\*** relation between the `Author` and the `Book` entities.

## Add Relation to The Book Entity

Open the `Books/Book.cs` in the `Acme.BookStore.Domain` project and add the following property to the `Book` entity:

````csharp

public Guid AuthorId { get; set; }

````

{{if DB=="EF"}}

> In this tutorial, we preferred to not add a **\*\*navigation property\*\*** to the `Author` entity from the `Book` class (like `public Author Author { get; set; }`). This is due to follow the DDD best practices (rule: refer to other aggregates only by id). However, you can add such a navigation property and configure it for the EF Core. In this way, you don't need to write join queries while getting books with their authors (like we will be doing below) which makes your application code simpler.

{{end}}

## Database & Data Migration

Added a new, required `AuthorId` property to the `Book` entity. But, **\*\*what about the existing books\*\*** on the database? They currently don't have `AuthorId`s and this will be a problem when we try to run the application.

This is a **\*\*typical migration problem\*\*** and the decision depends on your case;

\* If you haven't published your application to the production yet, you can just delete existing books in the database, or you can even delete the entire database in your development environment.

\* You can update the existing data programmatically on data migration or seed phase.

\* You can manually handle it on the database.

We prefer to **\*\*delete the database\*\*** {{if DB=="EF"}}(you can run the `Drop-Database` in the *\*Package Manager Console\**){{end}} since this is just an example project and data loss is not important. Since this topic is not related to the ABP Framework, we don't go deeper for all the scenarios.

{{if DB=="EF"}}

### Update the EF Core Mapping

Locate to `OnModelCreating` method in the `BookStoreDbContext` class that under the `EntityFrameworkCore` folder of the `Acme.BookStore.EntityFrameworkCore` project and change the `builder.Entity<Book>` part as shown below:

````csharp

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);

// ADD THE MAPPING FOR THE RELATION

b.HasOne<Author>().WithMany().HasForeignKey(x => x.AuthorId).IsRequired();

});

````

### Add New EF Core Migration

The startup solution is configured to use [Entity Framework Core Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/). Since we've changed the database mapping configuration, we should create a new migration and apply changes to the database.

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

````bash

dotnet ef migrations add Added\_AuthorId\_To\_Book

````

This should create a new migration class with the following code in its `Up` method:

````csharp

migrationBuilder.AddColumn<Guid>(

name: "AuthorId",

table: "AppBooks",

type: "uniqueidentifier",

nullable: false,

defaultValue: new Guid("00000000-0000-0000-0000-000000000000"));

migrationBuilder.CreateIndex(

name: "IX\_AppBooks\_AuthorId",

table: "AppBooks",

column: "AuthorId");

migrationBuilder.AddForeignKey(

name: "FK\_AppBooks\_AppAuthors\_AuthorId",

table: "AppBooks",

column: "AuthorId",

principalTable: "AppAuthors",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

````

\* Adds an `AuthorId` field to the `AppBooks` table.

\* Creates an index on the `AuthorId` field.

\* Declares the foreign key to the `AppAuthors` table.

> If you are using Visual Studio, you may want to use `Add-Migration Added\_AuthorId\_To\_Book -c BookStoreDbContext` and `Update-Database -Context BookStoreDbContext` commands in the *\*Package Manager Console (PMC)\**. In this case, ensure that {{if UI=="MVC"}}`Acme.BookStore.Web`{{else if UI=="BlazorServer"}}`Acme.BookStore.Blazor`{{else if UI=="Blazor" || UI=="NG"}}`Acme.BookStore.HttpApi.Host`{{end}} is the startup project and `Acme.BookStore.EntityFrameworkCore` is the *\*Default Project\** in PMC.

{{end}}

## Change the Data Seeder

Since the `AuthorId` is a required property of the `Book` entity, current data seeder code can not work. Open the `BookStoreDataSeederContributor` in the `Acme.BookStore.Domain` project and change as the following:

````csharp

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> \_bookRepository;

private readonly IAuthorRepository \_authorRepository;

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 context)

{

if (await \_bookRepository.GetCountAsync() > 0)

{

return;

}

var orwell = await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"George Orwell",

new DateTime(1903, 06, 25),

"Orwell produced literary criticism and poetry, fiction and polemical journalism; and is best known for the allegorical novella Animal Farm (1945) and the dystopian novel Nineteen Eighty-Four (1949)."

)

);

var douglas = await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"Douglas Adams",

new DateTime(1952, 03, 11),

"Douglas Adams was an English author, screenwriter, essayist, humorist, satirist and dramatist. Adams was an advocate for environmentalism and conservation, a lover of fast cars, technological innovation and the Apple Macintosh, and a self-proclaimed 'radical atheist'."

)

);

await \_bookRepository.InsertAsync(

new Book

{

AuthorId = orwell.Id, // SET THE AUTHOR

Name = "1984",

Type = BookType.Dystopia,

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

Price = 19.84f

},

autoSave: true

);

await \_bookRepository.InsertAsync(

new Book

{

AuthorId = douglas.Id, // SET THE AUTHOR

Name = "The Hitchhiker's Guide to the Galaxy",

Type = BookType.ScienceFiction,

PublishDate = new DateTime(1995, 9, 27),

Price = 42.0f

},

autoSave: true

);

}

}

````

The only change is that we set the `AuthorId` properties of the `Book` entities.

> Delete existing books or delete the database before executing the `DbMigrator`. See the *\*Database & Data Migration\** section above for more info.

{{if DB=="EF"}}

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

{{else if DB=="Mongo"}}

You can now run the `.DbMigrator` console application to **\*\*seed\*\*** the initial data.

{{end}}

## Application Layer

We will change the `BookAppService` to support the Author relation.

### Data Transfer Objects

Let's begin from the DTOs.

#### BookDto

Open the `BookDto` class in the `Books` folder of the `Acme.BookStore.Application.Contracts` project and add the following properties:

```csharp

public Guid AuthorId { get; set; }

public string AuthorName { get; set; }

```

The final `BookDto` class should be following:

```csharp

using System;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Books;

public class BookDto : AuditedEntityDto<Guid>

{

public Guid AuthorId { get; set; }

public string AuthorName { get; set; }

public string Name { get; set; }

public BookType Type { get; set; }

public DateTime PublishDate { get; set; }

public float Price { get; set; }

}

```

#### CreateUpdateBookDto

Open the `CreateUpdateBookDto` class in the `Books` folder of the `Acme.BookStore.Application.Contracts` project and add an `AuthorId` property as shown:

````csharp

public Guid AuthorId { get; set; }

````

#### AuthorLookupDto

Create a new class, `AuthorLookupDto`, inside the `Books` folder of the `Acme.BookStore.Application.Contracts` project:

````csharp

using System;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Books;

public class AuthorLookupDto : EntityDto<Guid>

{

public string Name { get; set; }

}

````

This will be used in a new method that will be added to the `IBookAppService`.

### IBookAppService

Open the `IBookAppService` interface in the `Books` folder of the `Acme.BookStore.Application.Contracts` project and add a new method, named `GetAuthorLookupAsync`, as shown below:

````csharp

using System;

using System.Threading.Tasks;

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

{

// ADD the NEW METHOD

Task<ListResultDto<AuthorLookupDto>> GetAuthorLookupAsync();

}

````

This new method will be used from the UI to get a list of authors and fill a dropdown list to select the author of a book.

### BookAppService

Open the `BookAppService` class in the `Books` folder of the `Acme.BookStore.Application` project and replace the file content with the following code:

{{if DB=="EF"}}

```csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Permissions;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Entities;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books;

[Authorize(BookStorePermissions.Books.Default)]

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

{

private readonly IAuthorRepository \_authorRepository;

public BookAppService(

IRepository<Book, Guid> repository,

IAuthorRepository authorRepository)

: base(repository)

{

\_authorRepository = authorRepository;

GetPolicyName = BookStorePermissions.Books.Default;

GetListPolicyName = BookStorePermissions.Books.Default;

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

public override async Task<BookDto> GetAsync(Guid id)

{

//Get the IQueryable<Book> from the repository

var queryable = await Repository.GetQueryableAsync();

//Prepare a query to join books and authors

var query = from book in queryable

join author in await \_authorRepository.GetQueryableAsync() on book.AuthorId equals author.Id

where book.Id == id

select new { book, author };

//Execute the query and get the book with author

var queryResult = await AsyncExecuter.FirstOrDefaultAsync(query);

if (queryResult == null)

{

throw new EntityNotFoundException(typeof(Book), id);

}

var bookDto = ObjectMapper.Map<Book, BookDto>(queryResult.book);

bookDto.AuthorName = queryResult.author.Name;

return bookDto;

}

public override async Task<PagedResultDto<BookDto>> GetListAsync(PagedAndSortedResultRequestDto input)

{

//Get the IQueryable<Book> from the repository

var queryable = await Repository.GetQueryableAsync();

//Prepare a query to join books and authors

var query = from book in queryable

join author in await \_authorRepository.GetQueryableAsync() on book.AuthorId equals author.Id

select new {book, author};

//Paging

query = query

.OrderBy(NormalizeSorting(input.Sorting))

.Skip(input.SkipCount)

.Take(input.MaxResultCount);

//Execute the query and get a list

var queryResult = await AsyncExecuter.ToListAsync(query);

//Convert the query result to a list of BookDto objects

var bookDtos = queryResult.Select(x =>

{

var bookDto = ObjectMapper.Map<Book, BookDto>(x.book);

bookDto.AuthorName = x.author.Name;

return bookDto;

}).ToList();

//Get the total count with another query

var totalCount = await Repository.GetCountAsync();

return new PagedResultDto<BookDto>(

totalCount,

bookDtos

);

}

public async Task<ListResultDto<AuthorLookupDto>> GetAuthorLookupAsync()

{

var authors = await \_authorRepository.GetListAsync();

return new ListResultDto<AuthorLookupDto>(

ObjectMapper.Map<List<Author>, List<AuthorLookupDto>>(authors)

);

}

private static string NormalizeSorting(string sorting)

{

if (sorting.IsNullOrEmpty())

{

return $"book.{nameof(Book.Name)}";

}

if (sorting.Contains("authorName", StringComparison.OrdinalIgnoreCase))

{

return sorting.Replace(

"authorName",

"author.Name",

StringComparison.OrdinalIgnoreCase

);

}

return $"book.{sorting}";

}

}

```

Let's see the changes we've done:

\* Added `[Authorize(BookStorePermissions.Books.Default)]` to authorize the methods we've newly added/overrode (remember, authorize attribute is valid for all the methods of the class when it is declared for a class).

\* Injected `IAuthorRepository` to query from the authors.

\* Overrode the `GetAsync` method of the base `CrudAppService`, which returns a single `BookDto` object with the given `id`.

\* Used a simple LINQ expression to join books and authors and query them together for the given book id.

\* Used `AsyncExecuter.FirstOrDefaultAsync(...)` to execute the query and get a result. It is a way to use asynchronous LINQ extensions without depending on the database provider API. Check the [repository documentation](../Repositories.md) to understand why we've used it.

\* Throws an `EntityNotFoundException` which results an `HTTP 404` (not found) result if requested book was not present in the database.

\* Finally, created a `BookDto` object using the `ObjectMapper`, then assigning the `AuthorName` manually.

\* Overrode the `GetListAsync` method of the base `CrudAppService`, which returns a list of books. The logic is similar to the previous method, so you can easily understand the code.

\* Created a new method: `GetAuthorLookupAsync`. This simple gets all the authors. The UI uses this method to fill a dropdown list and select and author while creating/editing books.

{{else if DB=="Mongo"}}

```csharp

using System;

using System.Collections.Generic;

using System.Linq.Dynamic.Core;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Permissions;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books;

[Authorize(BookStorePermissions.Books.Default)]

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

{

private readonly IAuthorRepository \_authorRepository;

public BookAppService(

IRepository<Book, Guid> repository,

IAuthorRepository authorRepository)

: base(repository)

{

\_authorRepository = authorRepository;

GetPolicyName = BookStorePermissions.Books.Default;

GetListPolicyName = BookStorePermissions.Books.Default;

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Create;

}

public async override Task<BookDto> GetAsync(Guid id)

{

var book = await Repository.GetAsync(id);

var bookDto = ObjectMapper.Map<Book, BookDto>(book);

var author = await \_authorRepository.GetAsync(book.AuthorId);

bookDto.AuthorName = author.Name;

return bookDto;

}

public async override Task<PagedResultDto<BookDto>>

GetListAsync(PagedAndSortedResultRequestDto input)

{

//Set a default sorting, if not provided

if (input.Sorting.IsNullOrWhiteSpace())

{

input.Sorting = nameof(Book.Name);

}

//Get the IQueryable<Book> from the repository

var queryable = await Repository.GetQueryableAsync();

//Get the books

var books = await AsyncExecuter.ToListAsync(

queryable

.OrderBy(input.Sorting)

.Skip(input.SkipCount)

.Take(input.MaxResultCount)

);

//Convert to DTOs

var bookDtos = ObjectMapper.Map<List<Book>, List<BookDto>>(books);

//Get a lookup dictionary for the related authors

var authorDictionary = await GetAuthorDictionaryAsync(books);

//Set AuthorName for the DTOs

bookDtos.ForEach(bookDto => bookDto.AuthorName =

authorDictionary[bookDto.AuthorId].Name);

//Get the total count with another query (required for the paging)

var totalCount = await Repository.GetCountAsync();

return new PagedResultDto<BookDto>(

totalCount,

bookDtos

);

}

public async Task<ListResultDto<AuthorLookupDto>> GetAuthorLookupAsync()

{

var authors = await \_authorRepository.GetListAsync();

return new ListResultDto<AuthorLookupDto>(

ObjectMapper.Map<List<Author>, List<AuthorLookupDto>>(authors)

);

}

private async Task<Dictionary<Guid, Author>>

GetAuthorDictionaryAsync(List<Book> books)

{

var authorIds = books

.Select(b => b.AuthorId)

.Distinct()

.ToArray();

var queryable = await \_authorRepository.GetQueryableAsync();

var authors = await AsyncExecuter.ToListAsync(

queryable.Where(a => authorIds.Contains(a.Id))

);

return authors.ToDictionary(x => x.Id, x => x);

}

}

```

Let's see the changes we've done:

\* Added `[Authorize(BookStorePermissions.Books.Default)]` to authorize the methods we've newly added/overrode (remember, authorize attribute is valid for all the methods of the class when it is declared for a class).

\* Injected `IAuthorRepository` to query from the authors.

\* Overrode the `GetAsync` method of the base `CrudAppService`, which returns a single `BookDto` object with the given `id`.

\* Overrode the `GetListAsync` method of the base `CrudAppService`, which returns a list of books. This code separately queries the authors from database and sets the name of the authors in the application code. Instead, you could create a custom repository method and perform a join query or take the power of the MongoDB API to get the books and their authors in a single query, which would be more performant.

\* Created a new method: `GetAuthorLookupAsync`. This simple gets all the authors. The UI uses this method to fill a dropdown list and select and author while creating/editing books.

{{end}}

### Object to Object Mapping Configuration

Introduced the `AuthorLookupDto` class and used object mapping inside the `GetAuthorLookupAsync` method. So, we need to add a new mapping definition inside the `BookStoreApplicationAutoMapperProfile.cs` file of the `Acme.BookStore.Application` project:

````csharp

CreateMap<Author, AuthorLookupDto>();

````

## Unit Tests

Some of the unit tests will fail since we made some changed on the `AuthorAppService`. Open the `BookAppService\_Tests` in the `Books` folder of the `Acme.BookStore.Application.Tests` project and change the content as the following:

```csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Shouldly;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Validation;

using Xunit;

namespace Acme.BookStore.Books;

{{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]{{end}}

public class BookAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IBookAppService \_bookAppService;

private readonly IAuthorAppService \_authorAppService;

public BookAppService\_Tests()

{

\_bookAppService = GetRequiredService<IBookAppService>();

\_authorAppService = GetRequiredService<IAuthorAppService>();

}

[Fact]

public async Task Should\_Get\_List\_Of\_Books()

{

//Act

var result = await \_bookAppService.GetListAsync(

new PagedAndSortedResultRequestDto()

);

//Assert

result.TotalCount.ShouldBeGreaterThan(0);

result.Items.ShouldContain(b => b.Name == "1984" &&

b.AuthorName == "George Orwell");

}

[Fact]

public async Task Should\_Create\_A\_Valid\_Book()

{

var authors = await \_authorAppService.GetListAsync(new GetAuthorListDto());

var firstAuthor = authors.Items.First();

//Act

var result = await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

AuthorId = firstAuthor.Id,

Name = "New test book 42",

Price = 10,

PublishDate = System.DateTime.Now,

Type = BookType.ScienceFiction

}

);

//Assert

result.Id.ShouldNotBe(Guid.Empty);

result.Name.ShouldBe("New test book 42");

}

[Fact]

public async Task Should\_Not\_Create\_A\_Book\_Without\_Name()

{

var exception = await Assert.ThrowsAsync<AbpValidationException>(async () =>

{

await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

});

exception.ValidationErrors

.ShouldContain(err => err.MemberNames.Any(m => m == "Name"));

}

}

```

\* Changed the assertion condition in the `Should\_Get\_List\_Of\_Books` from `b => b.Name == "1984"` to `b => b.Name == "1984" && b.AuthorName == "George Orwell"` to check if the author name was filled.

\* Changed the `Should\_Create\_A\_Valid\_Book` method to set the `AuthorId` while creating a new book, since it is required anymore.

## The User Interface

{{if UI=="MVC"}}

### The Book List

Book list page change is trivial. Open the `Pages/Books/Index.js` in the `Acme.BookStore.Web` project and add an `authorName` column between the `name` and `type` columns:

````js

...

{

title: l('Name'),

data: "name"

},

// ADDED the NEW AUTHOR NAME COLUMN

{

title: l('Author'),

data: "authorName"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType.' + data);

}

},

...

````

When you run the application, you can see the *\*Author\** column on the table:

![bookstore-added-author-to-book-list](images/bookstore-added-author-to-book-list-2.png)

### Create Modal

Open the `Pages/Books/CreateModal.cshtml.cs` in the `Acme.BookStore.Web` project and change the file content as shown below:

```csharp

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Books;

public class CreateModalModel : BookStorePageModel

{

[BindProperty]

public CreateBookViewModel Book { get; set; }

public List<SelectListItem> Authors { get; set; }

private readonly IBookAppService \_bookAppService;

public CreateModalModel(

IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public async Task OnGetAsync()

{

Book = new CreateBookViewModel();

var authorLookup = await \_bookAppService.GetAuthorLookupAsync();

Authors = authorLookup.Items

.Select(x => new SelectListItem(x.Name, x.Id.ToString()))

.ToList();

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.CreateAsync(

ObjectMapper.Map<CreateBookViewModel, CreateUpdateBookDto>(Book)

);

return NoContent();

}

public class CreateBookViewModel

{

[SelectItems(nameof(Authors))]

[DisplayName("Author")]

public Guid AuthorId { get; set; }

[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; }

}

}

```

\* Changed type of the `Book` property from `CreateUpdateBookDto` to the new `CreateBookViewModel` class defined in this file. The main motivation of this change to customize the model class based on the User Interface (UI) requirements. We didn't want to use UI-related `[SelectItems(nameof(Authors))]` and `[DisplayName("Author")]` attributes inside the `CreateUpdateBookDto` class.

\* Added `Authors` property that is filled inside the `OnGetAsync` method using the `IBookAppService.GetAuthorLookupAsync` method defined before.

\* Changed the `OnPostAsync` method to map `CreateBookViewModel` object to a `CreateUpdateBookDto` object since `IBookAppService.CreateAsync` expects a parameter of this type.

### Edit Modal

Open the `Pages/Books/EditModal.cshtml.cs` in the `Acme.BookStore.Web` project and change the file content as shown below:

```csharp

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Books;

public class EditModalModel : BookStorePageModel

{

[BindProperty]

public EditBookViewModel Book { get; set; }

public List<SelectListItem> Authors { get; set; }

private readonly IBookAppService \_bookAppService;

public EditModalModel(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public async Task OnGetAsync(Guid id)

{

var bookDto = await \_bookAppService.GetAsync(id);

Book = ObjectMapper.Map<BookDto, EditBookViewModel>(bookDto);

var authorLookup = await \_bookAppService.GetAuthorLookupAsync();

Authors = authorLookup.Items

.Select(x => new SelectListItem(x.Name, x.Id.ToString()))

.ToList();

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.UpdateAsync(

Book.Id,

ObjectMapper.Map<EditBookViewModel, CreateUpdateBookDto>(Book)

);

return NoContent();

}

public class EditBookViewModel

{

[HiddenInput]

public Guid Id { get; set; }

[SelectItems(nameof(Authors))]

[DisplayName("Author")]

public Guid AuthorId { get; set; }

[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; }

}

}

```

\* Changed type of the `Book` property from `CreateUpdateBookDto` to the new `EditBookViewModel` class defined in this file, just like done before for the create modal above.

\* Moved the `Id` property inside the new `EditBookViewModel` class.

\* Added `Authors` property that is filled inside the `OnGetAsync` method using the `IBookAppService.GetAuthorLookupAsync` method.

\* Changed the `OnPostAsync` method to map `EditBookViewModel` object to a `CreateUpdateBookDto` object since `IBookAppService.UpdateAsync` expects a parameter of this type.

These changes require a small change in the `EditModal.cshtml`. Remove the `<abp-input asp-for="Id" />` tag since we no longer need to it (since moved it to the `EditBookViewModel`). The final content of the `EditModal.cshtml` should be following:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model EditModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<abp-dynamic-form abp-model="Book" asp-page="/Books/EditModal">

<abp-modal>

<abp-modal-header title="@L["Update"].Value"></abp-modal-header>

<abp-modal-body>

<abp-form-content />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</abp-dynamic-form>

````

### Object to Object Mapping Configuration

The changes above requires to define some object to object mappings. Open the `BookStoreWebAutoMapperProfile.cs` in the `Acme.BookStore.Web` project and add the following mapping definitions inside the constructor:

```csharp

CreateMap<Pages.Books.CreateModalModel.CreateBookViewModel, CreateUpdateBookDto>();

CreateMap<BookDto, Pages.Books.EditModalModel.EditBookViewModel>();

CreateMap<Pages.Books.EditModalModel.EditBookViewModel, CreateUpdateBookDto>();

```

You can run the application and try to create a new book or update an existing book. You will see a drop down list on the create/update form to select the author of the book:

![bookstore-added-authors-to-modals](images/bookstore-added-authors-to-modals-2.png)

{{else if UI=="NG"}}

### Service Proxy Generation

Since the HTTP APIs have been changed, you need to update Angular client side [service proxies](../UI/Angular/Service-Proxies.md). Before running `generate-proxy` command, your host must be up and running.

Run the following command in the `angular` folder (you may need to stop the angular application):

```bash

abp generate-proxy -t ng

```

This command will update the service proxy files under the `/src/app/proxy/` folder.

### The Book List

Book list page change is trivial. Open the `/src/app/book/book.component.html` and add the following column definition between the `Name` and `Type` columns:

````html

<ngx-datatable-column

[name]="'::Author' | abpLocalization"

prop="authorName"

[sortable]="false"

></ngx-datatable-column>

````

When you run the application, you can see the *\*Author\** column on the table:

![bookstore-books-with-authorname-angular](images/bookstore-books-with-authorname-angular-2.png)

### Create/Edit Forms

The next step is to add an Author selection (dropdown) to the create/edit forms. The final UI will look like the one shown below:

![bookstore-angular-author-selection](images/bookstore-angular-author-selection-2.png)

Added the Author dropdown as the first element in the form.

Open the `/src/app/book/book.component.ts` and and change the content as shown below:

````js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions, AuthorLookupDto } from '@proxy/books';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

import { ConfirmationService, Confirmation } from '@abp/ng.theme.shared';

import { Observable } from 'rxjs';

import { map } from 'rxjs/operators';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService, { provide: NgbDateAdapter, useClass: NgbDateNativeAdapter }],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

form: FormGroup;

selectedBook = {} as BookDto;

authors$: Observable<AuthorLookupDto[]>;

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder,

private confirmation: ConfirmationService

) {

this.authors$ = bookService.getAuthorLookup().pipe(map((r) => r.items));

}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.selectedBook = {} as BookDto;

this.buildForm();

this.isModalOpen = true;

}

editBook(id: string) {

this.bookService.get(id).subscribe((book) => {

this.selectedBook = book;

this.buildForm();

this.isModalOpen = true;

});

}

buildForm() {

this.form = this.fb.group({

authorId: [this.selectedBook.authorId || null, Validators.required],

name: [this.selectedBook.name || null, Validators.required],

type: [this.selectedBook.type || null, Validators.required],

publishDate: [

this.selectedBook.publishDate ? new Date(this.selectedBook.publishDate) : null,

Validators.required,

],

price: [this.selectedBook.price || null, Validators.required],

});

}

save() {

if (this.form.invalid) {

return;

}

const request = this.selectedBook.id

? this.bookService.update(this.selectedBook.id, this.form.value)

: this.bookService.create(this.form.value);

request.subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

delete(id: string) {

this.confirmation.warn('::AreYouSureToDelete', 'AbpAccount::AreYouSure').subscribe((status) => {

if (status === Confirmation.Status.confirm) {

this.bookService.delete(id).subscribe(() => this.list.get());

}

});

}

}

````

\* Added imports for the `AuthorLookupDto`, `Observable` and `map`.

\* Added `authors$: Observable<AuthorLookupDto[]>;` field after the `selectedBook`.

\* Added `this.authors$ = bookService.getAuthorLookup().pipe(map((r) => r.items));` into the constructor.

\* Added ` authorId: [this.selectedBook.authorId || null, Validators.required],` into the `buildForm()` function.

Open the `/src/app/book/book.component.html` and add the following form group just before the book name form group:

````html

<div class="form-group">

<label for="author-id">Author</label><span> \* </span>

<select class="form-control" id="author-id" formControlName="authorId">

<option [ngValue]="null">Select author</option>

<option [ngValue]="author.id" \*ngFor="let author of authors$ | async">

{%{{{ author.name }}}%}

</option>

</select>

</div>

````

That's all. Just run the application and try to create or edit an author.

{{end}}

{{if UI == "Blazor" || UI == "BlazorServer"}}

### The Book List

It is very easy to show the *\*Author Name\** in the book list. Open the `/Pages/Books.razor` file in the `Acme.BookStore.Blazor` project and add the following `DataGridColumn` definition just after the `Name` (book name) column:

````xml

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.AuthorName)"

Caption="@L["Author"]"></DataGridColumn>

````

When you run the application, you can see the *\*Author\** column on the table:

![blazor-bookstore-book-list-with-authors](images/blazor-bookstore-book-list-with-authors-2.png)

### Create Book Modal

Add the following field to the `@code` section of the `Books.razor` file:

````csharp

IReadOnlyList<AuthorLookupDto> authorList = Array.Empty<AuthorLookupDto>();

````

Override the `OnInitializedAsync` method and adding the following code:

````csharp

protected override async Task OnInitializedAsync()

{

await base.OnInitializedAsync();

authorList = (await AppService.GetAuthorLookupAsync()).Items;

}

````

\* It is essential to call the `base.OnInitializedAsync()` since `AbpCrudPageBase` has some initialization code to be executed.

Override the `OpenCreateModalAsync` method and adding the following code:

````csharp

protected override async Task OpenCreateModalAsync()

{

if (!authorList.Any())

{

throw new UserFriendlyException(message: L["AnAuthorIsRequiredForCreatingBook"]);

}

await base.OpenCreateModalAsync();

NewEntity.AuthorId = authorList.First().Id;

}

````

The final `@code` block should be the following:

````csharp

@code

{

//ADDED A NEW FIELD

IReadOnlyList<AuthorLookupDto> authorList = Array.Empty<AuthorLookupDto>();

public Books() // Constructor

{

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

//GET AUTHORS ON INITIALIZATION

protected override async Task OnInitializedAsync()

{

await base.OnInitializedAsync();

authorList = (await AppService.GetAuthorLookupAsync()).Items;

}

protected override async Task OpenCreateModalAsync()

{

if (!authorList.Any())

{

throw new UserFriendlyException(message: L["AnAuthorIsRequiredForCreatingBook"]);

}

await base.OpenCreateModalAsync();

NewEntity.AuthorId = authorList.First().Id;

}

}

````

Finally, add the following `Field` definition into the `ModalBody` of the *\*Create\** modal, as the first item, before the `Name` field:

````xml

<Field>

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

<Select TValue="Guid" @bind-SelectedValue="@NewEntity.AuthorId">

@foreach (var author in authorList)

{

<SelectItem TValue="Guid" Value="@author.Id">

@author.Name

</SelectItem>

}

</Select>

</Field>

````

This requires to add a new localization key to the `en.json` file:

````js

"AnAuthorIsRequiredForCreatingBook": "An author is required to create a book"

````

You can run the application to see the *\*Author Selection\** while creating a new book:

![book-create-modal-with-author](images/book-create-modal-with-author-2.png)

### Edit Book Modal

Add the following `Field` definition into the `ModalBody` of the *\*Edit\** modal, as the first item, before the `Name` field:

````xml

<Field>

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

<Select TValue="Guid" @bind-SelectedValue="@EditingEntity.AuthorId">

@foreach (var author in authorList)

{

<SelectItem TValue="Guid" Value="@author.Id">

@author.Name

</SelectItem>

}

</Select>

</Field>

````

That's all. We are reusing the `authorList` defined for the *\*Create\** modal.

{{end}}

## Community Articles

https://community.abp.io/articles

## Migrating from the ASP.NET Boilerplate

# Migrating from ASP.NET Boilerplate to the ABP Framework

ABP Framework is **\*\*the successor\*\*** of the open source [ASP.NET Boilerplate](https://aspnetboilerplate.com/) framework. This guide aims to help you to **\*\*migrate your existing solutions\*\*** (you developed with the ASP.NET Boilerplate framework) to the ABP Framework.

## Introduction

**\*\*ASP.NET Boilerplate\*\*** is being **\*\*actively developed\*\*** [since 2013](https://github.com/aspnetboilerplate/aspnetboilerplate/graphs/contributors). It is loved, used and contributed by the community. It started as a side project of [a developer](http://halilibrahimkalkan.com/), but now it is officially maintained and improved by the company [Volosoft](https://volosoft.com/) in addition to the great community support.

ABP Framework has the same goal of the ASP.NET Boilerplate framework: **\*\*Don't Repeat Yourself\*\***! It provides infrastructure, tools and startup templates to make a developer's life easier while developing enterprise software solutions.

See [the introduction blog post](https://blog.abp.io/abp/Abp-vNext-Announcement) if you wonder why we needed to re-write the ASP.NET Boilerplate framework.

### Should I Migrate?

No, you don't have to!

\* ASP.NET Boilerplate is still in active development and maintenance.

\* It also works on the latest ASP.NET Core and related libraries and tools. It is up to date.

However, if you want to take the advantage of the new ABP Framework [features](https://abp.io/features) and the new architecture opportunities (like support for NoSQL databases, microservice compatibility, advanced modularity), you can use this document as a guide.

### What About the ASP.NET Zero?

[ASP.NET Zero](https://aspnetzero.com/) is a commercial product developed by the core ASP.NET Boilerplate team, on top of the ASP.NET Boilerplate framework. It provides pre-built application [features](https://aspnetzero.com/Features), code generation tooling and a nice looking modern UI. It is trusted and used by thousands of companies from all around the World.

We have created the [ABP Commercial](https://commercial.abp.io/) as an alternative to the ASP.NET Zero. ABP Commercial is more modular and upgradeable compared to the ASP.NET Zero. It currently has less features compared to ASP.NET Zero, but the gap will be closed by the time (it also has some features don't exist in the ASP.NET Zero).

We think ASP.NET Zero is still a good choice while starting a new application. It is production ready and mature solution delivered as a full source code. It is being actively developed and we are constantly adding new features.

We don't suggest to migrate your ASP.NET Zero based solution to the ABP Commercial if;

\* Your ASP.NET Zero solution is mature and it is in maintenance rather than a rapid development.

\* You don't have enough development time to perform the migration.

\* A monolithic solution fits in your business.

\* You've customized existing ASP.NET Zero features too much based on your requirements.

We also suggest you to compare the features of two products based on your needs.

If you have an ASP.NET Zero based solution and want to migrate to the ABP Commercial, this guide will also help you.

### ASP.NET MVC 5.x Projects

The ABP Framework doesn't support ASP.NET MVC 5.x, it only works with ASP.NET Core. So, if you migrate your ASP.NET MVC 5.x based projects, you will also deal with the .NET Core migration.

## The Migration Progress

We've designed the ABP Framework by **\*\*getting the best parts\*\*** of the ASP.NET Boilerplate framework, so it will be familiar to you if you've developed ASP.NET Boilerplate based applications.

In the ASP.NET Boilerplate, we have not worked much on the UI side, but used some free themes (we've used [metronic theme](https://keenthemes.com/metronic/) for ASP.NET Zero on the other side). In the ABP Framework, we worked a lot on the UI side (especially for the MVC / Razor Pages UI, because Angular already has a good modular system of its own). So, the **\*\*most challenging part\*\*** of the migration will be the **\*\*User Interface\*\*** of your solution.

ABP Framework is (and ASP.NET Boilerplate was) designed based on the [Domain Driven Design](https://docs.abp.io/en/abp/latest/Domain-Driven-Design) patterns & principles and the startup templates are layered based on the DDD layers. So, this guide respects to that layering model and explains the migration layer by layer.

## Creating the Solution

First step of the migration is to create a new solution. We suggest you to create a fresh new project using [the startup templates](https://abp.io/get-started) (see [this document](https://docs.abp.io/en/commercial/latest/getting-started) for the ABP Commercial).

After creating the project and running the application, you can copy your code from your existing solution to the new solution step by step, layer by layer.

### About Pre-Built Modules

The startup projects for the ABP Framework use the [pre-built modules](https://docs.abp.io/en/abp/latest/Modules/Index) (not all of them, but the essentials) and themes as NuGet/NPM packages. So, you don't see the source code of the modules/themes in your solution. This has an advantage that you can easily update these packages when a new version is released. However, you can not easily customize them as their source code in your hands.

We suggest to continue to use these modules as package references, in this way you can get new features easily (see [abp update command](https://docs.abp.io/en/abp/latest/CLI#update)). In this case, you have a few options to customize or extend the functionality of the used modules;

\* You can create your own entity and share the same database table with an entity in a used module. An example of this is the `AppUser` entity comes in the startup template.

\* You can [replace](https://docs.abp.io/en/abp/latest/Dependency-Injection#replace-a-service) a domain service, application service, controller, page model or other types of services with your own implementation. We suggest you to inherit from the existing implementation and override the method you need.

\* You can replace a `.cshtml` view, page, view component, partial view... with your own one using the [Virtual File System](https://docs.abp.io/en/abp/latest/Virtual-File-System).

\* You can override javascript, css, image or any other type of static files using the [Virtual File System](https://docs.abp.io/en/abp/latest/Virtual-File-System).

More extend/customization options will be developed and documented by the time. However, if you need to fully change the module implementation, it is best to add the [source code](https://github.com/abpframework/abp/tree/dev/modules) of the related module into your own solution and remove the package dependencies.

The source code of the modules and the themes are [MIT](https://opensource.org/licenses/MIT) licensed, you can fully own and customize it without any limitation (for the ABP Commercial, you can download the source code of a [module](https://commercial.abp.io/modules)/[theme](https://commercial.abp.io/themes) if you have a [license](https://commercial.abp.io/pricing) type that includes the source code).

## The Domain Layer

Most of your domain layer code will remain same, while you need to perform some minor changes in your domain objects.

### Aggregate Roots & Entities

The ABP Framework and the ASP.NET Boilerplate both have the `IEntity` and `IEntity<T>` interfaces and `Entity` and `Entity<T>` base classes to define entities but they have some differences.

If you have an entity in the ASP.NET Boilerplate application like that:

````csharp

public class Person : Entity //Default PK is int for the ASP.NET Boilerplate

{

...

}

````

Then your primary key (the `Id` property in the base class) is `int` which is the **\*\*default primary key\*\*** (PK) type for the ASP.NET Boilerplate. If you want to set another type of PK, you need to explicitly declare it:

````csharp

public class Person : Entity<Guid> //Set explicit PK in the ASP.NET Boilerplate

{

...

}

````

ABP Framework behaves differently and expects to **\*\*always explicitly set\*\*** the PK type:

````csharp

public class Person : Entity<Guid> //Set explicit PK in the ASP.NET Boilerplate

{

...

}

````

`Id` property (and the corresponding PK in the database) will be `Guid` in this case.

#### Composite Primary Keys

ABP Framework also has a non-generic `Entity` base class, but this time it has no `Id` property. Its purpose is to allow you to create entities with composite PKs. See [the documentation](https://docs.abp.io/en/abp/latest/Entities#entities-with-composite-keys) to learn more about the composite PKs.

#### Aggregate Root

It is best practice now to use the `AggregateRoot` base class instead of `Entity` for aggregate root entities. See [the documentation](https://docs.abp.io/en/abp/latest/Entities#aggregateroot-class) to learn more about the aggregate roots.

In opposite to the ASP.NET Boilerplate, the ABP Framework creates default repositories (`IRepository<T>`) **\*\*only for the aggregate roots\*\***. It doesn't create for other types derived from the `Entity`.

If you still want to create default repositories for all entity types, find the *\*YourProjectName\**EntityFrameworkCoreModule class in your solution and change `options.AddDefaultRepositories()` to `options.AddDefaultRepositories(includeAllEntities: true)` (it may be already like that for the application startup template).

#### Migrating the Existing Entities

We suggest & use the GUID as the PK type for all the ABP Framework modules. However, you can continue to use your existing PK types to migrate your database tables easier.

The challenging part will be the primary keys of the ASP.NET Boilerplate related entities, like Users, Roles, Tenants, Settings... etc. Our suggestion is to copy data from existing database to the new database tables using a tool or in a manual way (be careful about the foreign key values).

#### Documentation

See the documentation for details on the entities:

\* [ASP.NET Boilerplate - Entity documentation](https://aspnetboilerplate.com/Pages/Documents/Entities)

\* [ABP Framework - Entity documentation](https://docs.abp.io/en/abp/latest/Entities)

### Repositories

> ABP Framework creates default repositories (`IRepository<T>`) **\*\*only for the aggregate roots\*\***. It doesn't create for other types derived from the `Entity`. See the "Aggregate Root" section above for more information.

The ABP Framework and the ASP.NET Boilerplate both have the default generic repository system, but has some differences.

#### Injecting the Repositories

In the ASP.NET Boilerplate, there are two default repository interfaces you can directly inject and use:

\* `IRepository<TEntity>` (e.g. `IRepository<Person>`) is used for entities with `int` primary key (PK) which is the default PK type.

\* `IRepository<TEntity, TKey>` (e.g. `IRepository<Person, Guid>`) is used for entities with other types of PKs.

ABP Framework doesn't have a default PK type, so you need to **\*\*explicitly declare the PK type\*\*** of your entity, like `IRepository<Person, int>` or `IRepository<Person, Guid>`.

ABP Framework also has the `IRepository<TEntity>` (without PK), but it is mostly used when your entity has a composite PK (because this repository has no methods work with the `Id` property). See [the documentation](https://docs.abp.io/en/abp/latest/Entities#entities-with-composite-keys) to learn more about the **\*\*composite PKs\*\***.

#### Restricted Repositories

ABP Framework additionally provides a few repository interfaces:

\* `IBasicRepository<TEntity, TKey>` has the same methods with the `IRepository` except it doesn't have `IQueryable` support. It can be useful if you don't want to expose complex querying code to the application layer. In this case, you typically want to create custom repositories to encapsulate the querying logic. It is also useful for database providers those don't support `IQueryable`.

\* `IReadOnlyRepository<TEntity,TKey>` has the methods get data from the database, but doesn't contain any method change the database.

\* `IReadOnlyBasicRepository<TEntity, TKey>` is similar to the read only repository but also doesn't support `IQueryable`.

All the interfaces also have versions without `TKey` (like ``IReadOnlyRepository<TEntity>`) those can be used for composite PKs just like explained above.

#### GetAll() vs IQueryable

ASP.NET Boilerplate's repository has a `GetAll()` method that is used to obtain an `IQueryable` object to execute LINQ on it. An example application service calls the `GetAll()` method:

````csharp

public class PersonAppService : ApplicationService, IPersonAppService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task DoIt()

{

var people = await \_personRepository

.GetAll() //GetAll() returns IQueryable

.Where(p => p.BirthYear > 2000) //Use LINQ extension methods

.ToListAsync();

}

}

````

ABP Framework's repository have `GetQueryableAsync` instead:

````csharp

public class PersonAppService : ApplicationService, IPersonAppService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task DoIt()

{

var queryable = await \_personRepository.GetQueryableAsync();

var people = await queryable

.Where(p => p.BirthYear > 2000) //Use LINQ extension methods

.ToListAsync();

}

}

````

> Note that in order to use the async LINQ extension methods (like `ToListAsync` here), you may need to depend on the database provider (like EF Core) since these methods are defined in the database provider package, they are not standard LINQ methods. See the [repository document](Repositories.md) for alternative approaches for async query execution.

#### FirstOrDefault(predicate), Single()... Methods

ABP Framework repository has not such methods get predicate (expression) since the repository itself is `IQueryable` and all these methods are already standard LINQ extension methods those can be directly used.

However, it provides the following methods those can be used to query a single entity by its Id:

\* `FindAsync(id)` returns the entity or null if not found.

\* `GetAsync(id)` method returns the entity or throws an `EntityNotFoundException` (which causes HTTP 404 status code) if not found.

#### Sync vs Async

ABP Framework repository has no sync methods (like `Insert`). All the methods are async (like `InsertAsync`). So, if your application has sync repository method usages, convert them to async versions.

In general, ABP Framework forces you to completely use async everywhere, because mixing async & sync methods is not a recommended approach.

#### Documentation

See the documentation for details on the repositories:

\* [ASP.NET Boilerplate - Repository documentation](https://aspnetboilerplate.com/Pages/Documents/Repositories)

\* [ABP Framework - Repository documentation](https://docs.abp.io/en/abp/latest/Repositories)

### Domain Services

Your domain service logic mostly remains same on the migration. ABP Framework also defines the base `DomainService` class and the `IDomainService` interface just works like the ASP.NET Boilerplate.

## The Application Layer

Your application service logic remains similar on the migration. ABP Framework also defines the base `ApplicationService` class and the `IApplicationService` interface just works like the ASP.NET Boilerplate, but there are some differences in details.

### Declarative Authorization

ASP.NET Boilerplate has `AbpAuthorize` and `AbpMvcAuthorize` attributes for declarative authorization. Example usage:

````csharp

[AbpAuthorize("MyUserDeletionPermissionName")]

public async Task DeleteUserAsync(...)

{

...

}

````

ABP Framework doesn't has such a custom attribute. It uses the standard `Authorize` attribute in all layers.

````csharp

[Authorize("MyUserDeletionPermissionName")]

public async Task DeleteUserAsync(...)

{

...

}

````

This is possible with the better integration to the Microsoft Authorization Extensions libraries. See the Authorization section below for more information about the authorization system.

### CrudAppService and AsyncCrudAppService Classes

ASP.NET Boilerplate has `CrudAppService` (with sync service methods) and `AsyncCrudAppService` (with async service methods) classes.

ABP Framework only has the `CrudAppService` which actually has only the async methods (instead of sync methods).

ABP Framework's `CrudAppService` method signatures are slightly different than the old one. For example, old update method signature was ` Task<TEntityDto> UpdateAsync(TUpdateInput input) ` while the new one is ` Task<TGetOutputDto> UpdateAsync(TKey id, TUpdateInput input) `. The main difference is that it gets the Id of the updating entity as a separate parameter instead of including in the input DTO.

### Data Transfer Objects (DTOs)

There are similar base DTO classes (like `EntityDto`) in the ABP Framework too. So, you can find the corresponding DTO base class if you need.

#### Validation

You can continue to use the data annotation attributes to validate your DTOs just like in the ASP.NET Boilerplate.

ABP Framework doesn't include the ` ICustomValidate ` that does exists in the ASP.NET Boilerplate. Instead, you should implement the standard `IValidatableObject` interface for your custom validation logic.

## The Infrastructure Layer

### Namespaces

ASP.NET Boilerplate uses the `Abp.\*` namespaces while the ABP Framework uses the `Volo.Abp.\*` namespaces for the framework and pre-built fundamental modules.

In addition, there are also some pre-built application modules (like docs and blog modules) those are using the `Volo.\*` namespaces (like `Volo.Blogging.\*` and `Volo.Docs.\*`). We consider these modules as standalone open source products developed by Volosoft rather than add-ons or generic modules completing the ABP Framework and used in the applications. We've developed them as a module to make them re-usable as a part of a bigger solution.

### Module System

Both of the ASP.NET Boilerplate and the ABP Framework have the `AbpModule` while they are a bit different.

ASP.NET Boilerplate's `AbpModule` class has `PreInitialize`, `Initialize` and `PostInitialize` methods you can override and configure the framework and the depended modules. You can also register and resolve dependencies in these methods.

ABP Framework's `AbpModule` class has the `ConfigureServices` and `OnApplicationInitialization` methods (and their Pre and Post versions). It is similar to ASP.NET Core's Startup class. You configure other services and register dependencies in the `ConfigureServices`. However, you can now resolve dependencies in that point. You can resolve dependencies and configure the ASP.NET Core pipeline in the `OnApplicationInitialization` method while you can not register dependencies here. So, the new module classes separate dependency registration phase from dependency resolution phase since it follows the ASP.NET Core's approach.

### Dependency Injection

#### The DI Framework

ASP.NET Boilerplate is using the [Castle Windsor](http://www.castleproject.org/projects/windsor/) as the dependency injection framework. This is a fundamental dependency of the ASP.NET Boilerplate framework. We've got a lot of feedback to make the ASP.NET Boilerplate DI framework agnostic, but it was not so easy because of the design.

ABP Framework is dependency injection framework independent since it uses Microsoft's [Dependency Injection Extensions](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) library as an abstraction. None of the ABP Framework or module packages depends on any specific library.

However, ABP Framework doesn't use the Microsoft's base DI library because it has some missing features ABP Framework needs to: Property Injection and Interception. All the startup templates and the samples are using the [Autofac](https://autofac.org/) as the DI library and it is the only [officially integrated](Autofac-Integration.md) library to the ABP Framework. We suggest you to use the Autofac with the ABP Framework if you have not a good reason. If you have a good reason, please create an [issue](https://github.com/abpframework/abp/issues/new) on GitHub to request it or just implement it and send a pull request :)

#### Registering the Dependencies

Registering the dependencies are similar and mostly handled by the framework conventionally (like repositories, application services, controllers... etc). Implement the same `ITransientDependency`, `ISingletonDependency` and `IScopedDependency` interfaces for the services not registered by conventions.

When you need to manually register dependencies, use the `context.Services` in the `ConfigureServices` method of your module. Example:

````csharp

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Register an instance as singleton

context.Services.AddSingleton<TaxCalculator>(new TaxCalculator(taxRatio: 0.18));

//Register a factory method that resolves from IServiceProvider

context.Services.AddScoped<ITaxCalculator>(

sp => sp.GetRequiredService<TaxCalculator>()

);

}

}

````

See the ABP Framework [dependency injection document](https://docs.abp.io/en/abp/latest/Dependency-Injection) for details.

### Configuration vs Options System

ASP.NET Boilerplate has its own configuration system to configure the framework and the modules. For example, you could disable the audit logging in the `Initialize` method of your [module](https://aspnetboilerplate.com/Pages/Documents/Module-System):

````csharp

public override void Initialize()

{

Configuration.Auditing.IsEnabled = false;

}

````

ABP Framework uses [the options pattern](Options.md) to configure the framework and the modules. You typically configure the options in the `ConfigureServices` method of your [module](Module-Development-Basics.md):

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAuditingOptions>(options =>

{

options.IsEnabled = false;

});

}

````

Instead of a central configuration object, there are separated option classes for every module and feature those are defined in the related documents.

### IAbpSession vs ICurrentUser and ICurrentTenant

ASP.NET Boilerplate's `IAbpSession` service is used to obtain the current user and tenant information, like ` UserId ` and `TenantId`.

ABP Framework doesn't have the same service. Instead, use `ICurrentUser` and `ICurrentTenant` services. These services are defined as base properties in some common classes (like `ApplicationService` and `AbpController`), so you generally don't need to manually inject them. They also have much properties compared to the `IAbpSession`.

### Authorization

ABP Framework extends the [ASP.NET Core Authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction) by adding **\*\*permissions\*\*** as auto [policies](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies) and allowing the authorization system to be usable in the [application services](Application-Services.md) too.

#### AbpAuthorize vs Authorize

Use the standard `[Authorize]` and `[AllowAnonymous]` attributes instead of ASP.NET Boilerplate's custom `[AbpAuthorize]` and `[AbpAllowAnonymous]` attributes.

#### IPermissionChecker vs IAuthorizationService

Use the standard `IAuthorizationService` to check permissions instead of the ASP.NET Boilerplate's `IPermissionChecker` service. While `IPermissionChecker` also exists in the ABP Framework, it is used to explicitly use the permissions. Using `IAuthorizationService` is the recommended way since it covers other type of policy checks too.

#### AuthorizationProvider vs PermissionDefinitionProvider

You inherit from the `AuthorizationProvider` in the ASP.NET Boilerplate to define your permissions. ABP Framework replaces it by the `PermissionDefinitionProvider` base class. So, define your permissions by inheriting from the `PermissionDefinitionProvider` class.

### Unit of Work

Unit of work system has been designed to work seamlessly. For most of the cases, you don't need to change anything.

`UnitOfWork` attribute of the ABP Framework doesn't have the `ScopeOption` (type of `TransactionScopeOption`) property. Instead, use `IUnitOfWorkManager.Begin()` method with `requiresNew = true` to create an independent inner transaction in a transaction scope.

#### Data Filters

ASP.NET Boilerplate implements the data filtering system as a part of the unit of work. ABP Framework has a separate `IDataFilter` service.

See the [data filtering document](Data-Filtering.md) to learn how to enable/disable a filter.

See [the UOW documentation](Unit-Of-Work.md) for more about the UOW system.

### Multi-Tenancy

#### IMustHaveTenant & IMayHaveTenant vs IMultiTenant

ASP.NET Boilerplate defines `IMustHaveTenant` and `IMayHaveTenant` interfaces to implement them for your entities. In this way, your entities are automatically filtered according to the current tenant. Because of the design, there was a problem: You had to create a "Default" tenant in the database with "1" as the Id if you want to create a non multi-tenant application (this "Default" tenant was used as the single tenant).

ABP Framework has a single interface for multi-tenant entities: `IMultiTenant` which defines a nullable `TenantId` property of type `Guid`. If your application is not multi-tenant, then your entities will have null TenantId (instead of a default one).

On the migration, you need to change the TenantId field type and replace these interfaces with the `IMultiTenant`

#### Switch Between Tenants

In some cases you might need to switch to a tenant for a code scope and work with the tenant's data in this scope.

In ASP.NET Boilerplate, it is done using the `IUnitOfWorkManager` service:

````csharp

public async Task<List<Product>> GetProducts(int tenantId)

{

using (\_unitOfWorkManager.Current.SetTenantId(tenantId))

{

return await \_productRepository.GetAllListAsync();

}

}

````

In the ABP Framework it is done with the `ICurrentTenant` service:

````csharp

public async Task<List<Product>> GetProducts(Guid tenantId)

{

using (\_currentTenant.Change(tenantId))

{

return await \_productRepository.GetListAsync();

}

}

````

Pass `null` to the `Change` method to switch to the host side.

### Caching

ASP.NET Boilerplate has its [own distributed caching abstraction](https://aspnetboilerplate.com/Pages/Documents/Caching) which has in-memory and Redis implementations. You typically inject the `ICacheManager` service and use its `GetCache(...)` method to obtain a cache, then get and set objects in the cache.

ABP Framework uses and extends ASP.NET Core's [distributed caching abstraction](Caching.md). It defines the `IDistributedCache<T>` services to inject a cache and get/set objects.

### Logging

ASP.NET Boilerplate uses Castle Windsor's [logging facility](https://github.com/castleproject/Windsor/blob/master/docs/logging-facility.md) as an abstraction and supports multiple logging providers including Log4Net (the default one comes with the startup projects) and Serilog. You typically property-inject the logger:

````csharp

using Castle.Core.Logging; //1: Import Logging namespace

public class TaskAppService : ITaskAppService

{

//2: Getting a logger using property injection

public ILogger Logger { get; set; }

public TaskAppService()

{

//3: Do not write logs if no Logger supplied.

Logger = NullLogger.Instance;

}

public void CreateTask(CreateTaskInput input)

{

//4: Write logs

Logger.Info("Creating a new task with description: " + input.Description);

//...

}

}

````

ABP Framework depends on Microsoft's [logging extensions](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging) library which is also an abstraction and there are many providers implement it. Startup templates are using the Serilog as the pre-configured logging libary while it is easy to change in your project. The usage pattern is similar:

````csharp

//1: Import the Logging namespaces

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Logging.Abstractions;

public class TaskAppService : ITaskAppService

{

//2: Getting a logger using property injection

public ILogger<TaskAppService> Logger { get; set; }

public TaskAppService()

{

//3: Do not write logs if no Logger supplied.

Logger = NullLogger<TaskAppService>.Instance;

}

public void CreateTask(CreateTaskInput input)

{

//4: Write logs

Logger.Info("Creating a new task with description: " + input.Description);

//...

}

}

````

You inject the `ILogger<T>` instead of the `ILogger`.

### Object to Object Mapping

#### IObjectMapper Service

ASP.NET Boilerplate defines an `IObjectMapper` service ([see](https://aspnetboilerplate.com/Pages/Documents/Object-To-Object-Mapping)) and has an integration to the [AutoMapper](https://automapper.org/) library.

Example usage: Create a `User` object with the given `CreateUserInput` object:

````csharp

public void CreateUser(CreateUserInput input)

{

var user = ObjectMapper.Map<User>(input);

...

}

````

Example: Update an existing `User` properties with the given `UpdateUserInput` object:

````csharp

public async Task UpdateUserAsync(Guid id, UpdateUserInput input)

{

var user = await \_userRepository.GetAsync(id);

ObjectMapper.Map(input, user);

}

````

ABP Framework has the same `IObjectMapper` service ([see](Object-To-Object-Mapping.md)) and the AutoMapper integration with a slightly different mapping methods.

Example usage: Create a `User` object with the given `CreateUserInput` object:

````csharp

public void CreateUser(CreateUserInput input)

{

var user = ObjectMapper.Map<CreateUserInput, User>(input);

}

````

This time you need to explicitly declare the source type and target type (while ASP.NET Boilerplate was requiring only the target type).

Example: Update an existing `User` properties with the given `UpdateUserInput` object:

````csharp

public async Task UpdateUserAsync(Guid id, UpdateUserInput input)

{

var user = await \_userRepository.GetAsync(id);

ObjectMapper.Map<UpdateUserInput, User>(input, user);

}

````

Again, ABP Framework expects to explicitly set the source and target types.

#### AutoMapper Integration

##### Auto Mapping Attributes

ASP.NET Boilerplate has `AutoMapTo`, `AutoMapFrom` and `AutoMap` attributes to automatically create mappings for the declared types. Example:

````csharp

[AutoMapTo(typeof(User))]

public class CreateUserInput

{

public string Name { get; set; }

public string Surname { get; set; }

...

}

````

ABP Framework has no such attributes, because AutoMapper as a [similar attribute](https://automapper.readthedocs.io/en/latest/Attribute-mapping.html) now. You need to switch to AutoMapper's attribute.

##### Mapping Definitions

ABP Framework follows AutoMapper principles closely. You can define classes derived from the `Profile` class to define your mappings.

##### Configuration Validation

Configuration validation is a best practice for the AutoMapper to maintain your mapping configuration in a safe way.

See [the documentation](Object-To-Object-Mapping.md) for more information related to the object mapping.

### Setting Management

#### Defining the Settings

In an ASP.NET Boilerplate based application, you create a class deriving from the `SettingProvider` class, implement the `GetSettingDefinitions` method and add your class to the `Configuration.Settings.Providers` list.

In the ABP Framework, you need to derive your class from the `SettingDefinitionProvider` and implement the `Define` method. You don't need to register your class since the ABP Framework automatically discovers it.

#### Getting the Setting Values

ASP.NET Boilerplate provides the `ISettingManager` to read the setting values in the server side and `abp.setting.get(...)` method in the JavaScript side.

ABP Framework has the `ISettingProvider` service to read the setting values in the server side and `abp.setting.get(...)` method in the JavaScript side.

#### Setting the Setting Values

For ASP.NET Boilerplate, you use the same `ISettingManager` service to change the setting values.

ABP Framework separates it and provides the setting management module (pre-added to the startup projects) which has the ` ISettingManager ` to change the setting values. This separation was introduced to support tiered deployment scenarios (where `ISettingProvider` can also work in the client application while `ISettingManager ` can also work in the server (API) side).

### Clock

ASP.NET Boilerplate has a static `Clock` service ([see](https://aspnetboilerplate.com/Pages/Documents/Timing)) which is used to abstract the `DateTime` kind, so you can easily switch between Local and UTC times. You don't inject it, but just use the `Clock.Now` static method to obtain the current time.

ABP Framework has the `IClock` service ([see](Timing.md)) which has a similar goal, but now you need to inject it whenever you need it.

### Event Bus

ASP.NET Boilerplate has an in-process event bus system. You typically inject the `IEventBus` (or use the static instance `EventBus.Default`) to trigger an event. It automatically triggers events for entity changes (like `EntityCreatingEventData` and `EntityUpdatedEventData`). You create a class by implementing the `IEventHandler<T>` interface.

ABP Framework separates the event bus into two services: `ILocalEventBus` and `IDistributedEventBus`.

The local event bus is similar to the event bus of the ASP.NET Boilerplate while the distributed event bus is new feature introduced in the ABP Framework.

So, to migrate your code;

\* Use the `ILocalEventBus` instead of the `IEventBus`.

\* Implement the `ILocalEventHandler` instead of the `IEventHandler`.

> Note that ABP Framework has also an `IEventBus` interface, but it does exists to be a common interface for the local and distributed event bus. It is not injected and directly used.

### Feature Management

Feature system is used in multi-tenant applications to define features of your application check if given feature is available for the current tenant.

#### Defining Features

In the ASP.NET Boilerplate ([see](https://aspnetboilerplate.com/Pages/Documents/Feature-Management)), you create a class inheriting from the `FeatureProvider`, override the `SetFeatures` method and add your class to the `Configuration.Features.Providers` list.

In the ABP Framework ([see](Features.md)), you derive your class from the `FeatureDefinitionProvider` and override the `Define` method. No need to add your class to the configuration, it is automatically discovered by the framework.

#### Checking Features

You can continue to use the `RequiresFeature` attribute and `IFeatureChecker` service to check if a feature is enabled for the current tenant.

#### Changing the Feature Values

In the ABP Framework you use the `IFeatureManager` to change a feature value for a tenant.

### Audit Logging

The ASP.NET Boilerplate ([see](https://aspnetboilerplate.com/Pages/Documents/Audit-Logging)) and the ABP Framework ([see](Audit-Logging.md)) has similar audit logging systems. ABP Framework requires to add `UseAuditing()` middleware to the ASP.NET Core pipeline, which is already added in the startup templates. So, most of the times it will be work out of the box.

### Localization

ASP.NET Boilerplate supports XML and JSON files to define the localization key-values for the UI ([see](https://aspnetboilerplate.com/Pages/Documents/Localization)). ABP Framework only supports the JSON formatter localization files ([see](Localization.md)). So, you need to convert your XML file to JSON.

The ASP.NET Boilerplate has its own the `ILocalizationManager` service to be injected and used for the localization in the server side.

The ABP Framework uses [Microsoft localization extension](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization) library, so it is completely integrated to ASP.NET Core. You use the `IStringLocalizer<T>` service to get a localized text. Example:

````csharp

public class MyService

{

private readonly IStringLocalizer<TestResource> \_localizer;

public MyService(IStringLocalizer<TestResource> localizer)

{

\_localizer = localizer;

}

public void Foo()

{

var str = \_localizer["HelloWorld"]; //Get a localized text

}

}

````

So, you need to replace `ILocalizationManager` usage by the `IStringLocalizer`.

It also provides API used in the client side:

````js

var testResource = abp.localization.getResource('Test');

var str = testResource('HelloWorld');

````

It was like `abp.localization.localize(...)` in the ASP.NET Boilerplate.

### Navigation vs Menu

In ASP.NET Boilerplate you create a class deriving from the `NavigationProvider` to define your menu elements. Menu items has `requiredPermissionName` attributes to restrict access to a menu element. Menu items were static and your class is executed only one time.

In the ABP Framework you need to create a class implements the `IMenuContributor` interface. Your class is executed whenever the menu needs to be rendered. So, you can conditionally add menu items.

As an example, this is the menu contributor of the tenant management module:

````csharp

public class AbpTenantManagementWebMainMenuContributor : IMenuContributor

{

public async Task ConfigureMenuAsync(MenuConfigurationContext context)

{

//Add items only to the main menu

if (context.Menu.Name != StandardMenus.Main)

{

return;

}

//Get the standard administration menu item

var administrationMenu = context.Menu.GetAdministration();

//Resolve some needed services from the DI container

var l = context.GetLocalizer<AbpTenantManagementResource>();

var tenantManagementMenuItem = new ApplicationMenuItem(

TenantManagementMenuNames.GroupName,

l["Menu:TenantManagement"],

icon: "fa fa-users");

administrationMenu.AddItem(tenantManagementMenuItem);

//Conditionally add the "Tenants" menu item based on the permission

if (await context.IsGrantedAsync(TenantManagementPermissions.Tenants.Default))

{

tenantManagementMenuItem.AddItem(

new ApplicationMenuItem(

TenantManagementMenuNames.Tenants,

l["Tenants"],

url: "/TenantManagement/Tenants"));

}

}

}

````

So, you need to check permission using the `IAuthorizationService` if you want to show a menu item only when the user has the related permission.

> Navigation/Menu system is only for ASP.NET Core MVC / Razor Pages applications. Angular applications has a different system implemented in the startup templates.

## Missing Features

The following features are not present for the ABP Framework. Here, a list of some major missing features (and the related issue for that feature waiting on the ABP Framework GitHub repository):

\* [Multi-Lingual Entities](https://aspnetboilerplate.com/Pages/Documents/Multi-Lingual-Entities) ([#1754](https://github.com/abpframework/abp/issues/1754))

\* [Real time notification system](https://aspnetboilerplate.com/Pages/Documents/Notification-System) ([#633](https://github.com/abpframework/abp/issues/633))

\* [NHibernate Integration](https://aspnetboilerplate.com/Pages/Documents/NHibernate-Integration) ([#339](https://github.com/abpframework/abp/issues/339)) - We don't intent to work on this, but any community contribution welcome.

Some of these features will eventually be implemented. However, you can implement them yourself if they are important for you. If you want, you can [contribute](Contribution/Index.md) to the framework, it is appreciated.

# CLI

# ABP CLI

ABP CLI (Command Line Interface) is a command line tool to perform some common operations for ABP based solutions.

## Installation

ABP CLI is a [dotnet global tool](https://docs.microsoft.com/en-us/dotnet/core/tools/global-tools). Install it using a command line window:

````bash

dotnet tool install -g Volo.Abp.Cli

````

To update an existing installation:

````bash

dotnet tool update -g Volo.Abp.Cli

````

## Global Options

While each command may have a set of options, there are some global options that can be used with any command;

\* `--skip-cli-version-check`: Skips to check the latest version of the ABP CLI. If you don't specify, it will check the latest version and shows a warning message if there is a newer version of the ABP CLI.

## Commands

Here, is the list of all available commands before explaining their details:

\* **\*\***`help`**\*\***: Shows help on the usage of the ABP CLI.

\* **\*\***`cli`**\*\***: Update or remove ABP CLI.

\* **\*\***`new`**\*\***: Generates a new solution based on the ABP [startup templates](Startup-Templates/Index.md).

\* **\*\***`update`**\*\***: Automatically updates all ABP related NuGet and NPM packages in a solution.

\* **\*\***`clean`**\*\***: Deletes all `BIN` and `OBJ` folders in the current folder.

\* **\*\***`add-package`**\*\***: Adds an ABP package to a project.

\* **\*\***`add-module`**\*\***: Adds a [multi-package application module](https://docs.abp.io/en/abp/latest/Modules/Index) to a solution.

\* **\*\***`list-modules`**\*\***: Lists names of open-source application modules.

\* **\*\***`list-templates`**\*\***: Lists the names of available templates to create a solution.

\* **\*\***`get-source`**\*\***: Downloads the source code of a module.

\* **\*\***`generate-proxy`**\*\***: Generates client side proxies to use HTTP API endpoints.

\* **\*\***`remove-proxy`**\*\***: Removes previously generated client side proxies.

\* **\*\***`switch-to-preview`**\*\***: Switches to the latest preview version of the ABP Framework.

\* **\*\***`switch-to-nightly`**\*\***: Switches to the latest [nightly builds](Nightly-Builds.md) of the ABP related packages on a solution.

\* **\*\***`switch-to-stable`**\*\***: Switches to the latest stable versions of the ABP related packages on a solution.

\* **\*\***`switch-to-local`**\*\***: Changes NuGet package references on a solution to local project references.

\* **\*\***`translate`**\*\***: Simplifies to translate localization files when you have multiple JSON [localization](Localization.md) files in a source control repository.

\* **\*\***`login`**\*\***: Authenticates on your computer with your [abp.io](https://abp.io/) username and password.

\* **\*\***`login-info`**\*\***: Shows the current user's login information.

\* **\*\***`logout`**\*\***: Logouts from your computer if you've authenticated before.

\* **\*\***`bundle`**\*\***: Generates script and style references for ABP Blazor and MAUI Blazor project.

\* **\*\***`install-libs`**\*\***: Install NPM Packages for MVC / Razor Pages and Blazor Server UI types.

\* **\*\***`clear-download-cache`**\*\*** Clears the templates download cache.

### help

Shows basic usages of the ABP CLI.

Usage:

````bash

abp help [command-name]

````

Examples:

````bash

abp help # Shows a general help.

abp help new # Shows help about the "new" command.

````

### cli

Update or remove ABP CLI.

Usage:

````bash

abp cli [command-name]

````

Examples:

````bash

abp cli update

abp cli update --preview

abp cli update --version 5.0.0

abp cli remove

````

### new

Generates a new solution based on the ABP [startup templates](Startup-Templates/Index.md).

Usage:

````bash

abp new <solution-name> [options]

````

Example:

````bash

abp new Acme.BookStore

````

\* `Acme.BookStore` is the solution name here.

\* Common convention is to name a solution is like *\*YourCompany.YourProject\**. However, you can use different naming like *\*YourProject\** (single level namespacing) or *\*YourCompany.YourProduct.YourModule\** (three levels namespacing).

For more samples, go to [ABP CLI Create Solution Samples](CLI-New-Command-Samples.md)

#### Options

\* `--template` or `-t`: Specifies the template name. Default template name is `app`, which generates a web application. Available templates:

\* **\*\***`app`**\*\*** (default): [Application template](Startup-Templates/Application.md). Additional options:

\* `--ui` or `-u`: Specifies the UI framework. Default framework is `mvc`. Available frameworks:

\* `mvc`: ASP.NET Core MVC. There are some additional options for this template:

\* `--tiered`: Creates a tiered solution where Web and Http API layers are physically separated. If not specified, it creates a layered solution which is less complex and suitable for most scenarios.

\* `angular`: Angular UI. There are some additional options for this template:

\* `--separate-auth-server`: The Auth Server project comes as a separate project and runs at a different endpoint. It separates the Auth Server from the API Host application. If not specified, you will have a single endpoint in the server side.

\* `--pwa`: Specifies the project as Progressive Web Application.

\* `blazor`: Blazor UI. There are some additional options for this template:

\* `--separate-auth-server`The Auth Server project comes as a separate project and runs at a different endpoint. It separates the Auth Server from the API Host application. If not specified, you will have a single endpoint in the server side.

\* `--pwa`: Specifies the project as Progressive Web Application.

\* `blazor-server`: Blazor Server UI. There are some additional options for this template:

\* `--tiered`: The Auth Server and the API Host project comes as separate projects and run at different endpoints. It has 3 startup projects: *\*HttpApi.Host\**, *\*AuthServer\** and *\*Blazor\** and and each runs on different endpoints. If not specified, you will have a single endpoint for your web project.

\* `none`: Without UI. No front-end layer will be created. There are some additional options for this template:

\* `--separate-auth-server`: The Auth Server project comes as a separate project and runs at a different endpoint. It separates the Auth Server from the API Host application. If not specified, you will have a single endpoint in the server side.

\* `--mobile` or `-m`: Specifies the mobile application framework. If not specified, no mobile application will be created. Available options:

\* `react-native`: React Native.

\* `maui`: MAUI. This mobile option is only available for ABP Commercial.

\* `--database-provider` or `-d`: Specifies the database provider. Default provider is `ef`. Available providers:

\* `ef`: Entity Framework Core.

\* `mongodb`: MongoDB.

\* `--theme`: Specifes the theme. Default theme is `leptonx-lite`. Available themes:

\* `leptonx-lite`: [LeptonX Lite Theme](Themes/LeptonXLite/AspNetCore.md).

\* `basic`: [Basic Theme](UI/AspNetCore/Basic-Theme.md).

\* **\*\***`module`**\*\***: [Module template](Startup-Templates/Module.md). Additional options:

\* `--no-ui`: Specifies to not include the UI. This makes possible to create service-only modules (a.k.a. microservices - without UI).

\* **\*\***`console`**\*\***: [Console template](Startup-Templates/Console.md).

\* **\*\***`app-nolayers`**\*\***: [Single-layer application template](Startup-Templates/Application-Single-Layer.md). Additional options:

\* `--ui` or `-u`: Specifies the UI framework. Default framework is `mvc`. Available frameworks:

\* `mvc`: ASP.NET Core MVC.

\* `angular`: Angular UI.

\* `blazor`: Blazor UI.

\* `blazor-server`: Blazor Server UI.

\* `none`: Without UI.

\* `--database-provider` or `-d`: Specifies the database provider. Default provider is `ef`. Available providers:

\* `ef`: Entity Framework Core.

\* `mongodb`: MongoDB.

\* `--theme`: Specifes the theme. Default theme is `leptonx-lite`. Available themes:

\* `leptonx-lite`: [LeptonX Lite Theme](Themes/LeptonXLite/AspNetCore.md).

\* `basic`: [Basic Theme](UI/AspNetCore/Basic-Theme.md).

\* **\*\***`maui`**\*\***: .NET MAUI. A minimalist .NET MAUI application will be created if you specify this option.

\* `--output-folder` or `-o`: Specifies the output folder. Default value is the current directory.

\* `--version` or `-v`: Specifies the ABP & template version. It can be a [release tag](https://github.com/abpframework/abp/releases) or a [branch name](https://github.com/abpframework/abp/branches). Uses the latest release if not specified. Most of the times, you will want to use the latest version.

\* `--preview`: Use latest preview version.

\* `--template-source` or `-ts`: Specifies a custom template source to use to build the project. Local and network sources can be used(Like `D:\local-template` or `https://.../my-template-file.zip`).

\* `--create-solution-folder` or `-csf`: Specifies if the project will be in a new folder in the output folder or directly the output folder.

\* `--connection-string` or `-cs`: Overwrites the default connection strings in all `appsettings.json` files. The default connection string is `Server=localhost;Database=MyProjectName;Trusted\_Connection=True` for EF Core and it is configured to use the SQL Server. If you want to use the EF Core, but need to change the DBMS, you can change it as [described here](Entity-Framework-Core-Other-DBMS.md) (after creating the solution).

\* `--database-management-system` or `-dbms`: Sets the database management system. Default is **\*\*SQL Server\*\***. Supported DBMS's:

\* `SqlServer`

\* `MySQL`

\* `SQLite`

\* `Oracle`

\* `Oracle-Devart`

\* `PostgreSQL`

\* `--local-framework-ref --abp-path`: Uses local projects references to the ABP framework instead of using the NuGet packages. This can be useful if you download the ABP Framework source code and have a local reference to the framework from your application.

\* `--no-random-port`: Uses template's default ports.

\* `--skip-installing-libs` or `-sib`: Skip installing client side packages.

\* `--skip-cache` or `-sc`: Always download the latest from our server and refresh their templates folder cache.

\* `--with-public-website`: **\*\*Public Website\*\*** is a front-facing website for describing your project, listing your products and doing SEO for marketing purposes. Users can login and register on your website with this website.

See some [examples for the new command](CLI-New-Command-Samples.md) here.

### update

Updating all ABP related packages can be tedious since there are many packages of the framework and modules. This command automatically updates all ABP related NuGet and NPM packages in a solution or project to the latest versions. You can run it in the root folder of your solutions.

Usage:

````bash

abp update [options]

````

\* If you run in a directory with a .csproj file, it updates all ABP related packages of the project to the latest versions.

\* If you run in a directory with a .sln file, it updates all ABP related packages of the all projects of the solution to the latest versions.

\* If you run in a directory that contains multiple solutions in sub-folders, it can update all the solutions, including Angular projects.

Note that this command can upgrade your solution from a previous version, and also can upgrade it from a preview release to the stable release of the same version.

#### Options

\* `--npm`: Only updates NPM packages.

\* `--nuget`: Only updates NuGet packages.

\* `--solution-path` or `-sp`: Specify the solution path. Use the current directory by default

\* `--solution-name` or `-sn`: Specify the solution name. Search `\*.sln` files in the directory by default.

\* `--check-all`: Check the new version of each package separately. Default is `false`.

\* `--version` or `-v`: Specifies the version to use for update. If not specified, latest version is used.

### clean

Deletes all `BIN` and `OBJ` folders in the current folder.

Usage:

````bash

abp clean

````

### add-package

Adds an ABP package to a project by,

\* Adding related nuget package as a dependency to the project.

\* Adding `[DependsOn(...)]` attribute to the module class in the project (see the [module development document](Module-Development-Basics.md)).

> Notice that the added module may require additional configuration which is generally indicated in the documentation of the related package.

Basic usage:

````bash

abp add-package <package-name> [options]

````

Example:

````

abp add-package Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic

````

\* This example adds the `Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic` package to the project.

#### Options

\* `--project` or `-p`: Specifies the project (.csproj) file path. If not specified, CLI tries to find a .csproj file in the current directory.

\* `--with-source-code`: Downloads the source code of the package to your solution folder and uses local project references instead of NuGet/NPM packages.

\* `--add-to-solution-file`: Adds the downloaded package to your solution file, so you will also see the package when you open the solution on a IDE. (only available when `--with-source-code` is True)

> Currently only the source code of the basic theme packages([MVC](https://docs.abp.io/en/abp/latest/UI/AspNetCore/Basic-Theme) and [Blazor](https://docs.abp.io/en/abp/latest/UI/Blazor/Basic-Theme)) can be downloaded.

> - Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic

> - Volo.Abp.AspNetCore.Components.WebAssembly.BasicTheme

> - Volo.Abp.AspNetCore.Components.Web.BasicTheme

> - Volo.Abp.AspNetCore.Components.Server.BasicTheme

### add-module

Adds a [multi-package application module](Modules/Index) to a solution by finding all packages of the module, finding related projects in the solution and adding each package to the corresponding project in the solution.

It can also create a new module for your solution and add it to your solution. See `--new` option.

> A business module generally consists of several packages (because of layering, different database provider options or other reasons). Using `add-module` command dramatically simplifies adding a module to a solution. However, each module may require some additional configurations which is generally indicated in the documentation of the related module.

Usage:

````bash

abp add-module <module-name> [options]

````

Examples:

```bash

abp add-module Volo.Blogging

```

\* This example adds the `Volo.Blogging` module to the solution.

```bash

abp add-module ProductManagement --new --add-to-solution-file

```

\* This command creates a fresh new module customized for your solution (named `ProductManagement`) and adds it to your solution.

#### Options

\* `--solution` or `-s`: Specifies the solution (.sln) file path. If not specified, CLI tries to find a .sln file in the current directory.

\* `--skip-db-migrations`: For EF Core database provider, it automatically adds a new code first migration (`Add-Migration`) and updates the database (`Update-Database`) if necessary. Specify this option to skip this operation.

\* `-sp` or `--startup-project`: Relative path to the project folder of the startup project. Default value is the current folder.

\* `--new`: Creates a fresh new module (customized for your solution) and adds it to your solution.

\* `--with-source-code`: Downloads the source code of the module to your solution folder and uses local project references instead of NuGet/NPM packages. This options is always `True` if `--new` is used.

\* `--add-to-solution-file`: Adds the downloaded/created module to your solution file, so you will also see the projects of the module when you open the solution on a IDE. (only available when `--with-source-code` is `True`.)

### list-modules

Lists names of open-source application modules.

Usage:

````bash

abp list-modules [options]

````

Example:

```bash

abp list-modules

```

#### Options

\* `--include-pro-modules`: Includes commercial (pro) modules in the output.

### list-templates

Lists all available templates to create a solution.

Usage:

```bash

abp list-templates

```

### get-source

Downloads the source code of a module to your computer.

Usage:

````bash

abp get-source <module-name> [options]

````

Example:

```bash

abp get-source Volo.Blogging

abp get-source Volo.Blogging --local-framework-ref --abp-path D:\GitHub\abp

```

#### Options

\* `--output-folder` or `-o`: Specifies the directory that source code will be downloaded in. If not specified, current directory is used.

\* `--version` or `-v`: Specifies the version of the source code that will be downloaded. If not specified, latest version is used.

\* `--preview`: If no version option is specified, this option specifies if latest [preview version](Previews.md) will be used instead of latest stable version.

\* `--local-framework-ref --abp-path`: Path of [ABP Framework GitHub repository](https://github.com/abpframework/abp) in your computer. This will be used for converting project references to your local system. If this is not specified, project references will be converted to NuGet references.

### generate-proxy

Generates Angular, C# or JavaScript service proxies for your HTTP APIs to make easy to consume your services from the client side. Your host (server) application must be up and running before running this command.

Usage:

````bash

abp generate-proxy -t <client-type> [options]

````

Examples:

````bash

abp generate-proxy -t ng -url https://localhost:44302/

abp generate-proxy -t js -url https://localhost:44302/

abp generate-proxy -t csharp -url https://localhost:44302/

````

#### Options

\* `--type` or `-t`: The name of client type. Available clients:

\* `csharp`: C#, work in the `\*.HttpApi.Client` project directory. There are some additional options for this client:

\* `--without-contracts`: Avoid generating the application service interface, class, enum and dto types.

\* `--folder`: Folder name to place generated CSharp code in. Default value: `ClientProxies`.

\* `ng`: Angular. There are some additional options for this client:

\* `--api-name` or `-a`: The name of the API endpoint defined in the `/src/environments/environment.ts`. Default value: `default`.

\* `--source` or `-s`: Specifies the Angular project name to resolve the root namespace & API definition URL from. Default value: `defaultProject`.

\* `--target`: Specifies the Angular project name to place generated code in. Default value: `defaultProject`.

\* `--module`: Backend module name. Default value: `app`.

\* `--entry-point`: Targets the Angular project to place the generated code.

\* `--url`: Specifies api definition url. Default value is API Name's url in environment file.

\* `--prompt` or `-p`: Asks the options from the command line prompt (for the unspecified options).

\* `js`: JavaScript. work in the `\*.Web` project directory. There are some additional options for this client:

\* `--output` or `-o`: JavaScript file path or folder to place generated code in.

\* `--module` or `-m`: Specifies the name of the backend module you wish to generate proxies for. Default value: `app`.

\* `--working-directory` or `-wd`: Execution directory. For `csharp` and `js` client types.

\* `--url` or `-u`: API definition URL from.

\* `--service-type` or `-st`: Specifies the service type to generate. `application`, `integration` and `all`, Default value: `all` for C#, `application` for JavaScript / Angular.

> See the [Angular Service Proxies document](UI/Angular/Service-Proxies.md) for more.

### remove-proxy

Removes previously generated proxy code from the Angular, CSharp or JavaScript application. Your host (server) application must be up and running before running this command.

This can be especially useful when you generate proxies for multiple modules before and need to remove one of them later.

Usage:

````bash

abp remove-proxy -t <client-type> [options]

````

Examples:

````bash

abp remove-proxy -t ng

abp remove-proxy -t js -m identity -o Pages/Identity/client-proxies.js

abp remove-proxy -t csharp --folder MyProxies/InnerFolder

````

#### Options

\* `--type` or `-t`: The name of client type. Available clients:

\* `csharp`: C#, work in the `\*.HttpApi.Client` project directory. There are some additional options for this client:

\* `--folder`: Folder name to place generated CSharp code in. Default value: `ClientProxies`.

\* `ng`: Angular. There are some additional options for this client:

\* `--api-name` or `-a`: The name of the API endpoint defined in the `/src/environments/environment.ts`. Default value: `default`.

\* `--source` or `-s`: Specifies the Angular project name to resolve the root namespace & API definition URL from. Default value: `defaultProject`.

\* `--target`: Specifies the Angular project name to place generated code in. Default value: `defaultProject`.

\* `--url`: Specifies api definition url. Default value is API Name's url in environment file.

\* `--prompt` or `-p`: Asks the options from the command line prompt (for the unspecified options).

\* `js`: JavaScript. work in the `\*.Web` project directory. There are some additional options for this client:

\* `--output` or `-o`: JavaScript file path or folder to place generated code in.

\* `--module` or `-m`: Specifies the name of the backend module you wish to generate proxies for. Default value: `app`.

\* `--working-directory` or `-wd`: Execution directory. For `csharp` and `js` client types.

\* `--url` or `-u`: API definition URL from.

> See the [Angular Service Proxies document](UI/Angular/Service-Proxies.md) for more.

### switch-to-preview

You can use this command to switch your solution or project to latest preview version of the ABP framework.

Usage:

````bash

abp switch-to-preview [options]

````

#### Options

\* `--directory` or `-d`: Specifies the directory. The solution or project should be in that directory or in any of its sub directories. If not specified, default is the current directory.

### switch-to-nightly

You can use this command to switch your solution or project to latest [nightly](Nightly-Builds.md) preview version of the ABP framework packages.

Usage:

````bash

abp switch-to-nightly [options]

````

#### Options

\* `--directory` or `-d`: Specifies the directory. The solution or project should be in that directory or in any of its sub directories. If not specified, default is the current directory.

### switch-to-stable

If you're using the ABP Framework preview packages (including nightly previews), you can switch back to latest stable version using this command.

Usage:

````bash

abp switch-to-stable [options]

````

#### Options

\* `--directory` or `-d`: Specifies the directory. The solution or project should be in that directory or in any of its sub directories. If not specified, default is the current directory.

### switch-to-local

Changes all NuGet package references to local project references for all the .csproj files in the specified folder (and all its subfolders with any deep). It is not limited to ABP Framework or Module packages.

Usage:

````bash

abp switch-to-local [options]

````

#### Options

\* `--solution` or `-s`: Specifies the solution directory. The solution should be in that directory or in any of its sub directories. If not specified, default is the current directory.

\* `--paths` or `-p`: Specifies the local paths that the projects are inside. You can use `|` character to separate the paths.

Example:

````bash

abp switch-to-local --paths "D:\Github\abp|D:\Github\my-repo"

````

### translate

Simplifies to translate [localization](Localization.md) files when you have multiple JSON [localization](Localization.md) files in a source control repository.

\* This command will create a unified json file based on the reference culture.

\* It searches all the localization `JSON` files in the current directory and all subdirectories (recursively). Then creates a single file (named `abp-translation.json` by default) that includes all the entries need to be translated.

\* Once you translate the entries in this file, you can then apply your changes to the original localization files using the `--apply` command.

> The main purpose of this command is to translate ABP Framework localization files (since the [abp repository](https://github.com/abpframework/abp) has tens of localization files to be translated in different directories).

#### Creating the Translation File

First step is to create the unified translation file:

````bash

abp translate -c <culture> [options]

````

Example:

````bash

abp translate -c de

````

This command created the unified translation file for the `de` (German) culture.

##### Additional Options

\* `--reference-culture` or `-r`: Default `en`. Specifies the reference culture.

\* `--output` or `-o`: Output file name. Default `abp-translation.json`.

\* `--all-values` or `-all`: Include all keys to translate. By default, the unified translation file only includes the missing texts for the target culture. Specify this parameter if you may need to revise the values already translated before.

#### Applying Changes

Once you translate the entries in the unified translation file, you can apply your changes to the original localization files using the `--apply` parameter:

````bash

abp translate --apply # apply all changes

abp translate -a # shortcut for --apply

````

Then review changes on your source control system to be sure that it has changed the proper files and send a Pull Request if you've translated ABP Framework resources. Thank you in advance for your contribution.

##### Additional Options

\* `--file` or `-f`: Default: `abp-translation.json`. The translation file (use only if you've used the `--output` option before).

### login

Some features of the CLI requires to be logged in to abp.io platform. To login with your username write:

```bash

abp login <username> # Allows you to enter your password hidden

abp login <username> -p <password> # Specify the password as a parameter (password is visible)

abp login <username> --organization <organization> # If you have multiple organizations, you need set your active organization

abp login <username> -p <password> -o <organization> # You can enter both your password and organization in the same command

abp login <username> --device # Use device login flow

```

> When using the -p parameter, be careful as your password will be visible. It's useful for CI/CD automation pipelines.

A new login with an already active session overwrites the previous session.

### login-info

Shows your login information such as **\*\*Name\*\***, **\*\*Surname\*\***, **\*\*Username\*\***, **\*\*Email Address\*\*** and **\*\*Organization\*\***.

```bash

abp login-info

```

### logout

Logs you out by removing the session token from your computer.

```bash

abp logout

```

### bundle

This command generates script and style references for ABP Blazor WebAssembly and MAUI Blazor project and updates the **\*\*index.html\*\*** file. It helps developers to manage dependencies required by ABP modules easily. In order ```bundle``` command to work, its **\*\*executing directory\*\*** or passed ```--working-directory``` parameter's directory must contain a Blazor or MAUI Blazor project file(\*.csproj).

Usage:

````bash

abp bundle [options]

````

#### Options

\* ```--working-directory``` or ```-wd```: Specifies the working directory. This option is useful when executing directory doesn't contain a Blazor project file.

\* ```--force``` or ```-f```: Forces to build project before generating references.

\* ```--project-type``` or ```-t```: Specifies the project type. Default type is `webassembly`. Available types:

\* `webassembly`

\* `maui-blazor`

`bundle` command reads the `appsettings.json` file inside the Blazor and MAUI Blazor project for bundling options. For more details about managing style and script references in Blazor or MAUI Blazor apps, see [Managing Global Scripts & Styles](UI/Blazor/Global-Scripts-Styles.md)

### install-libs

This command install NPM Packages for MVC / Razor Pages and Blazor Server UI types. Its **\*\*executing directory\*\*** or passed ```--working-directory``` parameter's directory must contain a project file(\*.csproj).

`install-libs` command reads the `abp.resourcemapping.js` file to manage package. For more details see [Client Side Package Management](UI/AspNetCore/Client-Side-Package-Management.md).

Usage:

````bash

abp install-libs [options]

````

#### Options

\* ```--working-directory``` or ```-wd```: Specifies the working directory. This option is useful when executing directory doesn't contain a project file.

## See Also

\* [Examples for the new command](CLI-New-Command-Samples.md)

## Examples for the new command

# ABP CLI - New Solution Sample Commands

The `abp new` command creates an ABP solution or other artifacts based on an ABP template. [ABP CLI](CLI.md) has several parameters to create a new ABP solution. In this document we will show you some sample commands to create a new solution. All the project names are `Acme.BookStore`. Currently, the available mobile projects are `React Native` and `MAUI` mobile app. Available database providers are `Entity Framework Core` and `MongoDB`. All the commands starts with `abp new`.

## Angular

The following commands are for creating Angular UI projects:

\* **\*\*Entity Framework Core\*\***, no mobile app, creates the project in a new folder:

````bash

abp new Acme.BookStore -u angular --mobile none --database-provider ef -csf

````

\* **\*\*Entity Framework Core\*\***, default app template, **\*\*separate Auth Server\*\***, creates the project in a new folder:

```bash

abp new Acme.BookStore -t app -u angular -m none --separate-auth-server --database-provider ef -csf

```

\* **\*\*Entity Framework Core\*\***, **\*\*custom connection string\*\***, creates the project in a new folder:

```bash

abp new Acme.BookStore -u angular -csf --connection-string Server=localhost;Database=MyDatabase;Trusted\_Connection=True

```

\* **\*\*MongoDB\*\***, default app template, mobile project included, creates solution in `C:\MyProjects\Acme.BookStore`

```bash

abp new Acme.BookStore -u angular --database-provider mongodb --output-folder C:\MyProjects\Acme.BookStore

```

\* **\*\*MongoDB\*\***, default app template, no mobile app, **\*\*separate Auth Server\*\***, creates the project in a new folder:

```bash

abp new Acme.BookStore -t app -u angular -m none --separate-auth-server --database-provider mongodb -csf

```

## MVC

The following commands are for creating MVC UI projects:

\* **\*\*Entity Framework Core\*\***, no mobile app, creates the project in a new folder:

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef -csf

```

\* **\*\*Entity Framework Core\*\***, **\*\*tier architecture\*\*** (*\*Web and HTTP API are separated\**), no mobile app, creates the project in a new folder:

```bash

abp new Acme.BookStore -u mvc --mobile none --tiered --database-provider ef -csf

```

\* **\*\*MongoDB\*\***, no mobile app, creates the project in a new folder:

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider mongodb -csf

```

\* **\*\*MongoDB\*\***, **\*\*tier architecture\*\***, creates the project in a new folder:

```bash

abp new Acme.BookStore -u mvc --tiered --database-provider mongodb -csf

```

\* **\*\*Public Website\*\***, Entity Framework Core, no mobile app, creates the project in a new folder:

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef -csf --with-public-website

```

*\_Note that Public Website is only included in PRO templates.\_*

## Blazor WebAssembly

The following commands are for creating Blazor WASM projects:

\* **\*\*Entity Framework Core\*\***, no mobile app:

```bash

abp new Acme.BookStore -t app -u blazor --mobile none

```

\* **\*\*Entity Framework Core\*\***, **\*\*separate Auth Server\*\***, mobile app included:

```bash

abp new Acme.BookStore -u blazor --separate-auth-server

```

\* **\*\*MongoDB\*\***, no mobile app, creates the project in a new folder:

```bash

abp new Acme.BookStore -u blazor --database-provider mongodb --mobile none -csf

```

## Blazor Server

The following commands are for creating Blazor projects:

\* **\*\*Entity Framework Core\*\***, no mobile app:

```bash

abp new Acme.BookStore -t app -u blazor-server --mobile none

```

\* **\*\*Entity Framework Core\*\***, **\*\*separate Auth Server\*\***, **\*\*separate API Host\*\***, mobile app included:

```bash

abp new Acme.BookStore -u blazor-server --tiered

```

\* **\*\*MongoDB\*\***, no mobile app, creates the project in a new folder:

```bash

abp new Acme.BookStore -u blazor --database-provider mongodb --mobile none -csf

```

## No UI

In the default app template, there is always a frontend project. In this option there is no frontend project. It has a `HttpApi.Host` project to serve your HTTP WebAPIs. It's appropriate if you want to create a WebAPI service.

\* **\*\*Entity Framework Core\*\***, separate Auth Server, creates the project in a new folder:

```bash

abp new Acme.BookStore -u none --separate-auth-server -csf

```

\* **\*\*MongoDB\*\***, no mobile app:

```bash

abp new Acme.BookStore -u none --mobile none --database-provider mongodb

```

## Console application

It's a template of a basic .NET console application with ABP module architecture integrated. To create a console application use the following command:

\* This project consists of the following files: `Acme.BookStore.csproj`, `appsettings.json`, `BookStoreHostedService.cs`, `BookStoreModule.cs`, `HelloWorldService.cs` and `Program.cs`.

```bash

abp new Acme.BookStore -t console -csf

```

## Module

Module are reusable sub applications used by your main project. Using ABP Module is a best practice if you are building a microservice solution. As modules are not final applications, each module has all the frontend UI projects and database providers. The module template comes with an MVC UI to be able to develop without the final solution. But if you will develop your module under a final solution, you add `--no-ui` parameter to exclude MVC UI project.

\* Included frontends: `MVC`, `Angular`, `Blazor`. Included database providers: `Entity Framework Core`, `MongoDB`. Includes MVC startup project.

```bash

abp new Acme.IssueManagement -t module

```

\* The same with the upper but doesn't include MVC startup project.

```bash

abp new Acme.IssueManagement -t module --no-ui

```

\* Creates the module and adds it to your solution

```bash

abp new Acme.IssueManagement -t module --add-to-solution-file

```

## Create a solution from a specific version

When you create a solution, it always creates with the latest version. To create a project from an older version, you can pass the `--version` parameter.

\* Create a solution from v3.3.0, with Angular UI and Entity Framework Core.

```bash

abp new Acme.BookStore -t app -u angular -m none --database-provider ef -csf --version 3.3.0

```

To get the ABP version list, checkout following link: https://www.nuget.org/packages/Volo.Abp.Core/

## Create from a custom template

ABP CLI uses the default [app template](https://github.com/abpframework/abp/tree/dev/templates/app) to create your project. If you want to create a new solution from your customized template, you can use the parameter `--template-source`.

\* MVC UI, Entity Framework Core, no mobile app, using the template in `c:\MyProjects\templates\app` directory.

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef --template-source "c:\MyProjects\templates\app"

```

\* Same with the previous one except this command retrieves the template from the URL `https://myabp.com/app-template.zip`.

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef --template-source https://myabp.com/app-template.zip

```

## Create a preview version

ABP CLI always uses the latest version. In order to create a solution from a preview (RC) version add the `--preview` parameter.

\* Blazor UI, Entity Framework Core, no mobile, **\*\*preview version\*\***, creates the project in a new folder:

```bash

abp new Acme.BookStore -t app -u blazor --mobile none -csf --preview

```

## Choose database management system

The default database management system (DBMS) is `Entity Framework Core` / ` SQL Server`. You can choose a DBMS by passing `--database-management-system` parameter. [Accepted values](https://github.com/abpframework/abp/blob/dev/framework/src/Volo.Abp.Cli.Core/Volo/Abp/Cli/ProjectBuilding/Building/DatabaseManagementSystem.cs) are `SqlServer`, `MySQL`, `SQLite`, `Oracle`, `Oracle-Devart`, `PostgreSQL`. The default value is `SqlServer`.

\* Angular UI, **\*\*PostgreSQL\*\*** database, creates the project in a new folder:

```bash

abp new Acme.BookStore -u angular --database-management-system PostgreSQL -csf

```

## Use static HTTP ports

ABP CLI always assigns random ports to the hostable projects. If you need to keep the default ports and create a solution always with the same HTTP ports, add the parameter `--no-random-port`.

\* MVC UI, Entity Framework Core, **\*\*static ports\*\***, creates the project in a new folder:

```bash

abp new Acme.BookStore --no-random-port -csf

```

## Use local ABP framework references

ABP libraries are referenced from NuGet by default in the ABP solutions. Sometimes you need to reference ABP libraries locally to your solution. This is useful to debug the framework itself. Your local ABP Framework 's root directory must have the `Volo.Abp.sln` file. You can copy the content of the following directory to your file system https://github.com/abpframework/abp/tree/dev/framework

\* MVC UI, Entity Framework Core, **\*\*ABP libraries are local project references\*\***:

The local path must be the root directory of ABP repository.

If `C:\source\abp\framework\Volo.Abp.sln` is your framework solution path, then you must write `C:\source\abp` to the `--abp-path` paramter.

```bash

abp new Acme.BookStore --local-framework-ref --abp-path C:\source\abp

```

**\*\*Output\*\***:

As seen below, ABP Framework libraries are local project references.

```xml

<ItemGroup>

<ProjectReference Include="C:\source\abp\framework\src\Volo.Abp.Autofac\Volo.Abp.Autofac.csproj" />

<ProjectReference Include="C:\source\abp\framework\src\Volo.Abp.AspNetCore.Serilog\Volo.Abp.AspNetCore.Serilog.csproj" />

<ProjectReference Include="C:\source\abp\framework\src\Volo.Abp.AspNetCore.Authentication.JwtBearer\Volo.Abp.AspNetCore.Authentication.JwtBearer.csproj" />

<ProjectReference Include="..\Acme.BookStore.Application\Acme.BookStore.Application.csproj" />

<ProjectReference Include="..\Acme.BookStore.HttpApi\Acme.BookStore.HttpApi.csproj" />

<ProjectReference Include="..\Acme.BookStore.EntityFrameworkCore\Acme.BookStore.EntityFrameworkCore.csproj" />

</ItemGroup>

```

## See Also

\* [ABP CLI documentation](CLI.md)

# Startup Templates

## Overall

# Startup Templates

While you can start with an empty project and add needed packages manually, startup templates make easy and comfortable to start a new solution with the ABP framework. Click the name from the list below to see the documentation of the related startup template:

\* [\*\*`app`\*\*](Application.md): Application template.

\* [\*\*`app-nolayers`\*\*](Application-Single-Layer.md): Application (single layer) template.

\* [\*\*`module`\*\*](Module.md): Module/service template.

\* [\*\*`console`\*\*](Console.md): Console template.

\* [\*\*`WPF`\*\*](WPF.md): WPF template.

\* [\*\*`MAUI`\*\*](MAUI.md): MAUI template.

## Application

# Application Startup Template

## Introduction

This template provides a layered application structure based on the [Domain Driven Design](../Domain-Driven-Design.md) (DDD) practices.

This document explains **\*\*the solution structure\*\*** and projects in details. If you want to start quickly, follow the guides below:

\* [The getting started document](../Getting-Started.md) explains how to create a new application in a few minutes.

\* [The application development tutorial](../Tutorials/Part-1) explains step by step application development.

## How to Start With?

You can use the [ABP CLI](../CLI.md) to create a new project using this startup template. Alternatively, you can generate a CLI command from the [Get Started](https://abp.io/get-started) page. CLI approach is used here.

First, install the ABP CLI if you haven't installed it before:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then use the `abp new` command in an empty folder to create a new solution:

````bash

abp new Acme.BookStore -t app

````

\* `Acme.BookStore` is the solution name, like *\*YourCompany.YourProduct\**. You can use single-level, two-level or three-level naming.

\* This example specified the template name (`-t` or `--template` option). However, `app` is already the default template if you didn't specify it.

### Specify the UI Framework

This template provides multiple UI frameworks:

\* `mvc`: ASP.NET Core MVC UI with Razor Pages (default)

\* `blazor`: Blazor UI

\* `blazor-server`: Blazor Server UI

\* `angular`: Angular UI

Use the `-u` or `--ui` option to specify the UI framework:

````bash

abp new Acme.BookStore -u angular

````

### Specify the Database Provider

This template supports the following database providers:

- `ef`: Entity Framework Core (default)

- `mongodb`: MongoDB

Use `-d` (or `--database-provider`) option to specify the database provider:

````bash

abp new Acme.BookStore -d mongodb

````

### Specify the Mobile Application Framework

This template supports the following mobile application frameworks:

- `react-native`: React Native

Use the `-m` (or `--mobile`) option to specify the mobile application framework:

````bash

abp new Acme.BookStore -m react-native

````

If not specified, no mobile application will be created.

## Solution Structure

Based on the options you've specified, you will get a slightly different solution structure.

### Default Structure

If you don't specify any additional options, you will have a solution as shown below:

![bookstore-rider-solution-v6](../images/solution-structure-solution-explorer-rider.png)

Projects are organized in `src` and `test` folders. `src` folder contains the actual application which is layered based on [DDD](../Domain-Driven-Design.md) principles as mentioned before.

The diagram below shows the layers & project dependencies of the application:

![layered-project-dependencies](../images/layered-project-dependencies.png)

Each section below will explain the related project & its dependencies.

#### .Domain.Shared Project

This project contains constants, enums and other objects these are actually a part of the domain layer, but needed to be used by all layers/projects in the solution.

A `BookType` enum and a `BookConsts` class (which may have some constant fields for the `Book` entity, like `MaxNameLength`) are good candidates for this project.

\* This project has no dependency on other projects in the solution. All other projects depend on this one directly or indirectly.

#### .Domain Project

This is the domain layer of the solution. It mainly contains [entities, aggregate roots](../Entities.md), [domain services](../Domain-Services.md), [value objects](../Value-Objects.md), [repository interfaces](../Repositories.md) and other domain objects.

A `Book` entity, a `BookManager` domain service and an `IBookRepository` interface are good candidates for this project.

\* Depends on the `.Domain.Shared` because it uses constants, enums and other objects defined in that project.

#### .Application.Contracts Project

This project mainly contains [application service](../Application-Services.md) **\*\*interfaces\*\*** and [Data Transfer Objects](../Data-Transfer-Objects.md) (DTO) of the application layer. It exists to separate the interface & implementation of the application layer. In this way, the interface project can be shared to the clients as a contract package.

An `IBookAppService` interface and a `BookCreationDto` class are good candidates for this project.

\* Depends on the `.Domain.Shared` because it may use constants, enums and other shared objects of this project in the application service interfaces and DTOs.

#### .Application Project

This project contains the [application service](../Application-Services.md) **\*\*implementations\*\*** of the interfaces defined in the `.Application.Contracts` project.

A `BookAppService` class is a good candidate for this project.

\* Depends on the `.Application.Contracts` project to be able to implement the interfaces and use the DTOs.

\* Depends on the `.Domain` project to be able to use domain objects (entities, repository interfaces... etc.) to perform the application logic.

#### .EntityFrameworkCore Project

This is the integration project for the EF Core. It defines the `DbContext` and implements repository interfaces defined in the `.Domain` project.

\* Depends on the `.Domain` project to be able to reference to entities and repository interfaces.

> This project is available only if you are using EF Core as the database provider. If you select another database provider, its name will be different.

#### .DbMigrator Project

This is a console application that simplifies the execution of database migrations on development and production environments. When you run this application, it:

\* Creates the database if necessary.

\* Applies the pending database migrations.

\* Seeds initial data if needed.

> This project has its own `appsettings.json` file. So, if you want to change the database connection string, remember to change this file too.

Especially, seeding initial data is important at this point. ABP has a modular data seed infrastructure. See [its documentation](../Data-Seeding.md) for more about the data seeding.

While creating database & applying migrations seem only necessary for relational databases, this project comes even if you choose a NoSQL database provider (like MongoDB). In that case, it still seeds the initial data which is necessary for the application.

\* Depends on the `.EntityFrameworkCore` project (for EF Core) since it needs to access to the migrations.

\* Depends on the `.Application.Contracts` project to be able to access permission definitions, because the initial data seeder grants all permissions to the admin role by default.

#### .HttpApi Project

This project is used to define your API Controllers.

Most of the time you don't need to manually define API Controllers since ABP's [Auto API Controllers](../API/Auto-API-Controllers.md) feature creates them automagically based on your application layer. However, in case of you need to write API controllers, this is the best place to do it.

\* Depends on the `.Application.Contracts` project to be able to inject the application service interfaces.

#### .HttpApi.Client Project

This is a project that defines C# client proxies to use the HTTP APIs of the solution. You can share this library to 3rd-party clients, so they can easily consume your HTTP APIs in their Dotnet applications (For other types of applications, they can still use your APIs, either manually or using a tool in their own platform)

Most of the time you don't need to manually create C# client proxies, thanks to ABP's [Dynamic C# API Clients](../API/Dynamic-CSharp-API-Clients.md) feature.

`.HttpApi.Client.ConsoleTestApp` project is a console application created to demonstrate the usage of the client proxies.

\* Depends on the `.Application.Contracts` project to be able to share the same application service interfaces and DTOs with the remote service.

> You can delete this project & dependencies if you don't need to create C# client proxies for your APIs.

#### .Web Project

This project contains the User Interface (UI) of the application if you are using ASP.NET Core MVC UI. It contains Razor pages, JavaScript files, CSS files, images and so on...

This project contains the main `appsettings.json` file that contains the connection string and other configurations of the application.

\* Depends on the `.HttpApi` project since the UI layer needs to use APIs and the application service interfaces of the solution.

> If you check the source code of the `.Web.csproj` file, you will see the references to the `.Application` and the `.EntityFrameworkCore` projects.

>

> These references are actually not needed while coding your UI layer, because the UI layer normally doesn't depend on the EF Core or the Application layer's implementation. These startup templates are ready for tiered deployment, where the API layer is hosted on a separate server than the UI layer.

>

> However, if you don't choose the `--tiered` option, these references will be in the .Web project to be able to host the Web, API and application layers in a single application endpoint.

>

> This gives you the ability to use domain entities & repositories in your presentation layer. However, this is considered as a bad practice according to DDD.

#### Test Projects

The solution has multiple test projects, one for each layer:

\* `.Domain.Tests` is used to test the domain layer.

\* `.Application.Tests` is used to test the application layer.

\* `.EntityFrameworkCore.Tests` is used to test EF Core configuration and custom repositories.

\* `.Web.Tests` is used to test the UI (if you are using ASP.NET Core MVC UI).

\* `.TestBase` is a base (shared) project for all tests.

In addition, `.HttpApi.Client.ConsoleTestApp` is a console application (not an automated test project) which demonstrate the usage of HTTP APIs from a .NET application.

Test projects are prepared for integration testing;

\* It is fully integrated into the ABP framework and all services in your application.

\* It uses SQLite in-memory database for EF Core. For MongoDB, it uses the [Mongo2Go](https://github.com/Mongo2Go/Mongo2Go) library.

\* Authorization is disabled, so any application service can be easily used in tests.

You can still create unit tests for your classes which will be harder to write (because you will need to prepare mock/fake objects), but faster to run (because it only tests a single class and skips all the initialization processes).

#### How to Run?

Set `.Web` as the startup project and run the application. The default username is `admin` and the password is `1q2w3E\*`.

See [Getting Started With the ASP.NET Core MVC Template](../Getting-Started-AspNetCore-MVC-Template.md) for more information.

### Tiered Structure

If you have selected the ASP.NET Core UI and specified the `--tiered` option, the solution created will be a tiered solution. The purpose of the tiered structure is to be able to **\*\*deploy Web applications and HTTP API to different servers\*\***:

![bookstore-visual-studio-solution-v3](../images/tiered-solution-servers.png)

\* Browser runs your UI by executing HTML, CSS & JavaScript.

\* Web servers host static UI files (CSS, JavaScript, image... etc.) & dynamic components (e.g. Razor pages). It performs HTTP requests to the API server to execute the business logic of the application.

\* The API Server hosts the HTTP APIs which then use the application & domain layers of the application to perform the business logic.

\* Finally, database server hosts your database.

So, the resulting solution allows a 4-tiered deployment, by comparing to 3-tiered deployment of the default structure explained before.

> Unless you actually need such a 4-tiered deployment, it's suggested to go with the default structure which is simpler to develop, deploy and maintain.

The solution structure is shown below:

![bookstore-rider-solution-v6](../images/bookstore-rider-solution-tiered.png)

As different from the default structure, two new projects come into play: `.AuthServer` & `.HttpApi.Host`.

#### .AuthServer Project

This project is used as an authentication server for other projects. `.Web` project uses OpenId Connect Authentication to get identity and access tokens for the current user from the AuthServer. Then uses the access token to call the HTTP API server. HTTP API server uses bearer token authentication to obtain claims from the access token to authorize the current user.

![tiered-solution-applications](../images/tiered-solution-applications-authserver.png)

ABP uses the [OpenIddict Module](../Modules/OpenIddict.md) that uses the open-source [OpenIddict-core](https://github.com/openiddict/openiddict-core) library for the authentication between applications. See [OpenIddict documentation](https://documentation.openiddict.com/) for details about the OpenIddict and OpenID Connect protocol.

It has its own `appsettings.json` that contains database connection and other configurations.

#### .HttpApi.Host Project

This project is an application that hosts the API of the solution. It has its own `appsettings.json` that contains database connection and other configurations.

#### .Web Project

Just like the default structure, this project contains the User Interface (UI) of the application. It contains razor pages, JavaScript files, style files, images and so on...

This project contains an `appsettings.json` file, but this time it does not have a connection string because it never connects to the database. Instead, it mainly contains the endpoint of the remote API server and the authentication server.

#### Pre-requirements

\* [Redis](https://redis.io/): The applications use Redis as a distributed cache. So, you need to have Redis installed & running.

#### How to Run?

You should run the application with the given order:

\* First, run the `.AuthServer` since other applications depend on it.

\* Then run the `.HttpApi.Host` since it is used by the `.Web` application.

\* Finally, you can run the `.Web` project and login to the application (using `admin` as the username and `1q2w3E\*` as the password).

### Blazor UI

If you choose `Blazor` as the UI Framework (using the `-u blazor` or `-u blazor-server` option), the solution will have a project named `.Blazor`. This project contains the Blazor UI application. According to your choice, it will be a Blazor WebAssembly or Blazor Server application. If Blazor WebAssembly is selected, the solution will also have a `.HttpApi.Host`. This project is an ASP.NET Core application that hosts the backend application for the Blazor single page application.

#### .Blazor Project (Server)

The Blazor Server project is similar to the ASP.NET Core MVC project. It replaces `.Web` project with `.Blazor` in the solution structure above. It has the same folder structure and the same application flow. Since it's an ASP.NET Core application, it can contain **\*\*.cshtml\*\*** files and **\*\*.razor\*\*** components at the same time. If routing matches a razor component, the Blazor UI will be used. Otherwise, the request will be handled by the MVC framework.

![abp solution structure blazor server](../images/layered-project-dependencies-blazor-server.png)

#### .Blazor Project (WebAssembly)

The Blazor WebAssembly project is a single page application that runs on the browser. You'll see it as `.Blazor` project in the solution. It uses the `.HttpApi.Host` project to communicate with the backend. It can't be used without the backend application. It contains only **\*\*.razor\*\*** components. It's a pure client-side application. It doesn't have any server-side code. Everything in this layer will be for the client side.

![abp solution structure blazor wasm](../images/layered-project-dependencies-blazor-wasm.png)

### Angular UI

If you choose `Angular` as the UI framework (using the `-u angular` option), the solution is being separated into two folders:

\* `angular` folder contains the Angular UI application, the client-side code.

\* `aspnet-core` folder contains the ASP.NET Core solution, the server-side code.

The server-side is similar to the solution described above. `\*.HttpApi.Host` project serves the API, so the `Angular` application consumes it.

Angular application folder structure looks like below:

![angular-folder-structure](../images/angular-folder-structure.png)

Each of ABP Commercial modules is an NPM package. Some ABP modules are added as a dependency in `package.json`. These modules install with their dependencies. To see all ABP packages, you can run the following command in the `angular` folder:

```bash

yarn list --pattern abp

```

Angular application module structure:

![Angular template structure diagram](../images/angular-template-structure-diagram.png)

#### AppModule

`AppModule` is the root module of the application. Some of the ABP modules and some essential modules are imported to `AppModule`.

ABP Config modules have also been imported to `AppModule` for initial requirements of the lazy-loadable ABP modules.

#### AppRoutingModule

There are lazy-loadable ABP modules in the `AppRoutingModule` as routes.

> Paths of ABP Modules should not be changed.

You should add `routes` property in the `data` object to add a link on the menu to redirect to your custom pages.

```js

{

path: 'dashboard',

loadChildren: () => import('./dashboard/dashboard.module').then(m => m.DashboardModule),

canActivate: [AuthGuard, PermissionGuard],

data: {

routes: {

name: 'ProjectName::Menu:Dashboard',

order: 2,

iconClass: 'fa fa-dashboard',

requiredPolicy: 'ProjectName.Dashboard.Host'

} as ABP.Route

}

}

```

In the above example;

\* If the user is not logged in, AuthGuard blocks access and redirects to the login page.

\* PermissionGuard checks the user's permission with the `requiredPolicy` property of the `routes` object. If the user is not authorized to access the page, the 403 page appears.

\* The `name` property of `routes` is the menu link label. A localization key can be defined.

\* The `iconClass` property of the `routes` object is the menu link icon class.

\* The `requiredPolicy` property of the `routes` object is the required policy key to access the page.

After the above `routes` definition, if the user is authorized, the dashboard link will appear on the menu.

#### Shared Module

The modules that may be required for all modules have been imported to the `SharedModule`. You should import `SharedModule` to all modules.

See the [Sharing Modules](https://angular.io/guide/sharing-ngmodules) document.

#### Environments

The files under the `src/environments` folder have the essential configuration of the application.

#### Home Module

Home module is an example lazy-loadable module that loads on the root address of the application.

#### Styles

The required style files are added to the `styles` array in `angular.json`. `AppComponent` loads some style files lazily via `LazyLoadService` after the main bundle is loaded to shorten the first rendering time.

#### Testing

You should create your tests in the same folder as the file you want to test.

See the [testing document](https://angular.io/guide/testing).

#### Depended Packages

\* [NG Bootstrap](https://ng-bootstrap.github.io/) is used as UI component library.

\* [NGXS](https://www.ngxs.io/) is used as state management library.

\* [angular-oauth2-oidc](https://github.com/manfredsteyer/angular-oauth2-oidc) is used to support for OAuth 2 and OpenId Connect (OIDC).

\* [Chart.js](https://www.chartjs.org/) is used to create widgets.

\* [ngx-validate](https://github.com/ng-turkey/ngx-validate) is used for dynamic validation of reactive forms.

### React Native

If the `-m react-native` option is specified in the new project command, the solution includes the [React Native](https://reactnative.dev/) application in the `react-native` folder.

The server-side is similar to the solution described above. `\*.HttpApi.Host` project serves the API, so the React Native application consumes it.

The React Native application was generated with [Expo](https://expo.io/). Expo is a set of tools built around React Native to help you quickly start an app and, while it has many features.

React Native application folder structure as like below:

![react-native-folder-structure](../images/react-native-folder-structure.png)

\* `App.js` is the bootstrap component of the application.

\* `Environment.js` file has the essential configuration of the application. `prod` and `dev` configurations are defined in this file.

\* [Contexts](https://reactjs.org/docs/context.html) are created in the `src/contexts` folder.

\* [Higher order components](https://reactjs.org/docs/higher-order-components.html) are created in the `src/hocs` folder.

\* [Custom hooks](https://reactjs.org/docs/hooks-custom.html#extracting-a-custom-hook) are created in `src/hooks`.

\* [Axios interceptors](https://github.com/axios/axios#interceptors) are created in the `src/interceptors` folder.

\* Utility functions are exported from `src/utils` folder.

#### Components

Components that can be used on all screens are created in the `src/components` folder. All components have been created as a function that is able to use [hooks](https://reactjs.org/docs/hooks-intro.html).

#### Screens

![react-native-navigation-structure](../images/react-native-navigation-structure.png)

Screens are created by creating folders that separate their names in the `src/screens` folder. Certain parts of some screens can be split into components.

Each screen is used in a navigator in the `src/navigators` folder.

#### Navigation

[React Navigation](https://reactnavigation.org/) is used as a navigation library. Navigators are created in the `src/navigators`. A [drawer](https://reactnavigation.org/docs/drawer-based-navigation/) navigator and several [stack](https://reactnavigation.org/docs/hello-react-navigation/#installing-the-stack-navigator-library) navigators have been created in this folder. See the [above diagram](#screens) for the navigation structure.

#### State Management

[Redux](https://redux.js.org/) is used as a state management library. [Redux Toolkit](https://redux-toolkit.js.org/) library is used as a toolset for efficient Redux development.

Actions, reducers, sagas and selectors are created in the `src/store` folder. Store folder is as below:

![react-native-store-folder](../images/react-native-store-folder.png)

\* [\*\*Store\*\*](https://redux.js.org/basics/store) is defined in the `src/store/index.js` file.

\* [\*\*Actions\*\*](https://redux.js.org/basics/actions/) are payloads of information that send data from your application to your store.

\* [\*\*Reducers\*\*](https://redux.js.org/basics/reducers) specify how the application's state changes in response to actions sent to the store.

\* [\*\*Redux-Saga\*\*](https://redux-saga.js.org/) is a library that aims to make application side effects (i.e. asynchronous things like data fetching and impure things like accessing the browser cache) easier to manage. Sagas are created in the `src/store/sagas` folder.

\* [\*\*Reselect\*\*](https://github.com/reduxjs/reselect) library is used to create memoized selectors. Selectors are created in the `src/store/selectors` folder.

#### APIs

[Axios](https://github.com/axios/axios) is used as an HTTP client library. An Axios instance has exported from `src/api/API.js` file to make HTTP calls with the same config. `src/api` folder also has the API files that have been created for API calls.

#### Theming

[Native Base](https://nativebase.io/) is used as UI components library. Native Base components can customize easily. See the [Native Base customize](https://docs.nativebase.io/customizing-components) documentation. We followed the same way.

\* Native Base theme variables are in the `src/theme/variables` folder.

\* Native Base component styles are in the `src/theme/components` folder. These files have been generated with Native Base's `ejectTheme` script.

\* Styles of components override with the files under the `src/theme/overrides` folder.

#### Testing

Unit tests will be created.

See the [Testing Overview](https://reactjs.org/docs/testing.html) document.

#### Depended Libraries

\* [Native Base](https://nativebase.io/) is used as UI components library.

\* [React Navigation](https://reactnavigation.org/) is used as navigation library.

\* [Axios](https://github.com/axios/axios) is used as an HTTP client library.

\* [Redux](https://redux.js.org/) is used as state management library.

\* [Redux Toolkit](https://redux-toolkit.js.org/) library is used as a toolset for efficient Redux development.

\* [Redux-Saga](https://redux-saga.js.org/) is used to manage asynchronous processes.

\* [Redux Persist](https://github.com/rt2zz/redux-persist) is used as state persistence.

\* [Reselect](https://github.com/reduxjs/reselect) is used to create memoized selectors.

\* [i18n-js](https://github.com/fnando/i18n-js) is used as i18n library.

\* [expo-font](https://docs.expo.io/versions/latest/sdk/font/) library allows loading fonts easily.

\* [Formik](https://github.com/jaredpalmer/formik) is used to build forms.

\* [Yup](https://github.com/jquense/yup) is used for form validations.

## Social / External Logins

If you want to configure social/external logins for your application, please follow the [Social/External Logins](../Authentication/Social-External-Logins.md) document.

## What's Next?

- [The getting started document](../Getting-Started.md) explains how to create a new application in a few minutes.

- [The application development tutorial](../Tutorials/Part-1.md) explains step by step application development.

## Application (Single Layer)

# Application (Single Layer) Startup Template

## Introduction

This template provides a simple solution structure with a single project. This document explains that solution structure in details.

### The Difference Between the Application Startup Templates

ABP's [Application Startup Template](Application.md) provides a well-organized and layered solution to create maintainable business applications based on the [Domain Driven Design](../Domain-Driven-Design.md) (DDD) practices. However, some developers find this template a little bit complex for simple and short-term applications. The single-layer application template has been created to provide a simpler development model for such applications. This template has the same functionality, features and modules on runtime with the [Application Startup Template](Application.md) but the development model is minimal and everything is in a single project (`.csproj`).

## How to Start with It?

You can use the [ABP CLI](../CLI.md) to create a new project using this startup template. Alternatively, you can generate a CLI command for this startup template from the [Get Started](https://abp.io/get-started) page. In this section, we will use the ABP CLI.

Firstly, install the ABP CLI if you haven't installed it before:

```bash

dotnet tool install -g Volo.Abp.Cli

```

Then, use the `abp new` command in an empty folder to create a new solution:

```bash

abp new Acme.BookStore -t app-nolayers

```

\* `Acme.BookStore` is the solution name, like *\*YourCompany.YourProduct\**. You can use single-level, two-level or three-level naming.

\* In this example, the `-t` (or `--template`) option specifies the template name.

### Specify the UI Framework

This template provides multiple UI frameworks:

\* `mvc`: ASP.NET Core MVC UI with Razor Pages (default)

\* `blazor`: Blazor UI

\* `blazor-server`: Blazor Server UI

\* `angular`: Angular UI

\* `none`: Without UI (for HTTP API development)

Use the `-u` (or `--ui`) option to specify the UI framework while creating the solution:

```bash

abp new Acme.BookStore -t app-nolayers -u angular

```

This example specifies the UI type (the `-u` option) as `angular`. You can also specify `mvc`, `blazor`, `blazor-server` or `none` for the UI type.

### Specify the Database Provider

This template supports the following database providers:

- `ef`: Entity Framework Core (default)

- `mongodb`: MongoDB

Use the `-d` (or `--database-provider`) option to specify the database provider while creating the solution:

```bash

abp new Acme.BookStore -t app-nolayers -d mongodb

```

## Solution Structure

If you don't specify any additional options while creating an `app-nolayers` template, you will have a solution as shown below:

![](../images/bookstore-single-layer-solution-structure.png)

In the next sections, we will explain the structure based on this example. Your startup solution can be slightly different based on your preferences.

### Folder Structure

Since this template provides a single-project solution, we've separated concerns into folders instead of projects. You can see the pre-defined folders as shown below:

![](../images/single-layer-folder-structure.png)

\* Define your database mappings (for [EF Core](../Entity-Framework-Core.md) or [MongoDB](../MongoDB.md)) and [repositories](../Repositories.md) in the `Data` folder.

\* Define your [entities](../Entities.md) in the `Entities` folder.

\* Define your UI localization keys/values in the `Localization` folder.

\* Define your UI menu items in the `Menus` folder.

\* Define your [object-to-object mapping](../Object-To-Object-Mapping.md) classes in the `ObjectMapping` folder.

\* Define your UI pages (Razor Pages) in the `Pages` folder (create `Controllers` and `Views` folder yourself if you prefer the MVC pattern).

\* Define your [application services](../Application-Services.md) in the `Services` folder.

### How to Run?

Before running the application, you need to create the database and seed the initial data. To do that, you can run the following command in the directory of your project (in the same folder of the `.csproj` file):

```bash

dotnet run --migrate-database

```

This command will create the database and seed the initial data for you. Then you can run the application with any IDE that supports .NET or by running the `dotnet run` command in the directory of your project. The default username is `admin` and the password is `1q2w3E\*`.

> While creating a database & applying migrations seem only necessary for relational databases, you should run this command even if you choose a NoSQL database provider (like MongoDB). In that case, it still seeds the initial data which is necessary for the application.

### The Angular UI

If you choose `Angular` as the UI framework, the solution will be separated into two folders:

\* An `angular` folder that contains the Angular UI application, the client-side code.

\* An `aspnet-core` folder that contains the ASP.NET Core solution (a single project), the server-side code.

The server-side is similar to the solution described in the *\*Solution Structure\** section above. This project serves the API, so the Angular application can consume it.

The client-side application consumes the HTTP APIs as mentioned. You can see the folder structure of the Angular project shown below:

![](../images/single-layer-angular-folder-structure.png)

## Module

# Module Startup Template

This template can be used to create a **\*\*reusable [**application module**](../Modules/Index.md)\*\*** based on the [module development best practices & conventions](../Best-Practices/Index.md). It is also suitable for creating **\*\*microservices\*\*** (with or without UI).

## How to Start With?

You can use the [ABP CLI](../CLI.md) to create a new project using this startup template. Alternatively, you can generate a CLI command from the [Get Started](https://abp.io/get-started) page. CLI approach is used here.

First, install the ABP CLI if you haven't installed before:

```bash

dotnet tool install -g Volo.Abp.Cli

```

Then use the `abp new` command in an empty folder to create a new solution:

```bash

abp new Acme.IssueManagement -t module

```

- `Acme.IssueManagement` is the solution name, like *\*YourCompany.YourProduct\**. You can use single level, two-levels or three-levels naming.

### Without User Interface

The template comes with MVC, Blazor & Angular user interfaces by default. You can use `--no-ui` option to not include any of these UI layers.

````bash

abp new Acme.IssueManagement -t module --no-ui

````

## Solution Structure

Based on the options you've specified, you will get a slightly different solution structure. If you don't specify any option, you will have a solution like shown below:

![issuemanagement-module-solution](../images/issuemanagement-module-solution.png)

Projects are organized as `src`, `test` and `host` folders:

\* `src` folder contains the actual module which is layered based on [DDD](../Domain-Driven-Design.md) principles.

\* `test` folder contains unit & integration tests.

\* `host` folder contains applications with different configurations to demonstrate how to host the module in an application. These are not a part of the module, but useful on development.

The diagram below shows the layers & project dependencies of the module:

![layered-project-dependencies-module](../images/layered-project-dependencies-module.png)

Each section below will explain the related project & its dependencies.

### .Domain.Shared Project

This project contains constants, enums and other objects these are actually a part of the domain layer, but needed to be used by all layers/projects in the solution.

An `IssueType` enum and an `IssueConsts` class (which may have some constant fields for the `Issue` entity, like `MaxTitleLength`) are good candidates for this project.

- This project has no dependency to other projects in the solution. All other projects depend on this directly or indirectly.

### .Domain Project

This is the domain layer of the solution. It mainly contains [entities, aggregate roots](../Entities.md), [domain services](../Domain-Services.md), value types, [repository interfaces](../Repositories.md) and other domain objects.

An `Issue` entity, an `IssueManager` domain service and an `IIssueRepository` interface are good candidates for this project.

- Depends on the `.Domain.Shared` because it uses constants, enums and other objects defined in that project.

### .Application.Contracts Project

This project mainly contains [application service](../Application-Services.md) **\*\*interfaces\*\*** and [Data Transfer Objects](../Data-Transfer-Objects.md) (DTO) of the application layer. It does exists to separate interface & implementation of the application layer. In this way, the interface project can be shared to the clients as a contract package.

An `IIssueAppService` interface and an `IssueCreationDto` class are good candidates for this project.

- Depends on the `.Domain.Shared` because it may use constants, enums and other shared objects of this project in the application service interfaces and DTOs.

### .Application Project

This project contains the [application service](../Application-Services.md) **\*\*implementations\*\*** of the interfaces defined in the `.Application.Contracts` project.

An `IssueAppService` class is a good candidate for this project.

- Depends on the `.Application.Contracts` project to be able to implement the interfaces and use the DTOs.

- Depends on the `.Domain` project to be able to use domain objects (entities, repository interfaces... etc.) to perform the application logic.

### .EntityFrameworkCore Project

This is the integration project for EF Core. It defines the `DbContext` and implements repository interfaces defined in the `.Domain` project.

- Depends on the `.Domain` project to be able to reference to entities and repository interfaces.

> You can delete this project if you don't want to support EF Core for your module.

### .MongoDB Project

This is the integration project for MongoDB.

- Depends on the `.Domain` project to be able to reference to entities and repository interfaces.

> You can delete this project if you don't want to support MongoDB for your module.

### Test Projects

The solution has multiple test projects, one for each layer:

- `.Domain.Tests` is used to test the domain layer.

- `.Application.Tests` is used to test the application layer.

- `.EntityFrameworkCore.Tests` is used to test EF Core configuration and custom repositories.

- `.MongoDB.Tests` is used to test MongoDB configuration and custom repositories.

- `.TestBase` is a base (shared) project for all tests.

In addition, `.HttpApi.Client.ConsoleTestApp` is a console application (not an automated test project) which demonstrate the usage of HTTP APIs from a Dotnet application.

Test projects are prepared for integration testing;

- It is fully integrated to ABP framework and all services in your application.

- It uses SQLite in-memory database for EF Core. For MongoDB, it uses the [Mongo2Go](https://github.com/Mongo2Go/Mongo2Go) library.

- Authorization is disabled, so any application service can be easily used in tests.

You can still create unit tests for your classes which will be harder to write (because you will need to prepare mock/fake objects), but faster to run (because it only tests a single class and skips all initialization process).

> Domain & Application tests are using EF Core. If you remove EF Core integration or you want to use MongoDB for testing these layers, you should manually change project references & module dependencies.

### Host Projects

The solution has a few host applications to run your module. Host applications are used to run your module in a fully configured application. It is useful on development. Host applications includes some other modules in addition to the module being developed:

Host applications support two types of scenarios.

#### Single (Unified) Application Scenario

If your module has a UI, then `.Web.Unified` application is used to host the UI and API on a single point. It has its own `appsettings.json` file (that includes the database connection string) and EF Core database migrations.

For the `.Web.Unified` application, there is a single database, named `YourProjectName\_Unified` (like *\*IssueManagement\_Unified\** for this sample).

> If you've selected the `--no-ui` option, this project will not be in your solution.

##### How to Run?

Set `host/YourProjectName.Web.Unified` as the startup project, run `Update-Database` command for the EF Core from Package Manager Console and run your application. Default username is `admin` and password is `1q2w3E\*`.

#### Separated Deployment & Databases Scenario

In this scenario, there are three applications;

\* `.AuthServer` application is an authentication server used by other applications. It has its own `appsettings.json` that contains database connection and other configurations.

\* `.HttpApi.Host` hosts the HTTP API of the module. It has its own `appsettings.json` that contains database connections and other configurations.

\* `.Web.Host` host the UI of the module. This project contains an `appsettings.json` file, but it does not have a connection string because it never connects to the database. Instead, it mainly contains endpoint of the remote API server and the authentication server.

The diagram below shows the relation of the applications:

![tiered-solution-applications](../images/tiered-solution-applications.png)

`.Web.Host` project uses OpenId Connect Authentication to get identity and access tokens for the current user from the `.AuthServer`. Then uses the access token to call the `.HttpApi.Host`. HTTP API server uses bearer token authentication to obtain claims from the access token to authorize the current user.

##### Pre-requirements

\* [Redis](https://redis.io/): The applications use Redis as as distributed cache. So, you need to have Redis installed & running.

##### How to Run?

You should run the application with the given order:

- First, run the `.AuthServer` since other applications depends on it.

- Then run the `.HttpApi.Host` since it is used by the `.Web.Host` application.

- Finally, you can run the `.Web.Host` project and login to the application using `admin` as the username and `1q2w3E\*` as the password.

## UI

### Angular UI

The solution will have a folder called `angular` in it. This is where the Angular client-side code is located. When you open that folder in an IDE, the folder structure will look like below:

![Folder structure of ABP Angular module project](../images/angular-module-folder-structure.png)

\* *\_angular/projects/issue-management\_* folder contains the Angular module project.

\* *\_angular/projects/dev-app\_* folder contains a development application that runs your module.

The server-side is similar to the solution described above. `\*.HttpApi.Host` project serves the API and the `Angular` demo application consumes it. You will not need to run the `.Web.Host` project though.

#### How to Run the Angular Development App

For module development, you will need the `dev-app` project up and running. So, here is how we can start the development server.

First, we need to install dependencies:

1. Open your terminal at the root folder, i.e. `angular`.

2. Run `yarn` or `npm install`.

The dependencies will be installed and some of them are ABP modules published as NPM packages. To see all ABP packages, you can run the following command in the `angular` folder:

```bash

yarn list --pattern abp

```

> There is no equivalent of this command in npm.

The module you will develop depends on two of these ABP packages: *\_@abp/ng.core\_* and *\_@abp/ng.theme.shared\_*. Rest of the ABP modules are included in *\_package.json\_* because of the `dev-app` project.

Once all dependencies are installed, follow the steps below to serve your development app:

1. Make sure `.AuthServer` and `\*.HttpApi.Host` projects are up and running.

2. Open your terminal at the root folder, i.e. `angular`.

3. Run `yarn start` or `npm start`.

![ABP Angular module dev-app project](../images/angular-module-dev-app-project.png)

The issue management page is empty in the beginning. You may change the content in `IssueManagementComponent` at the *\_angular/projects/issue-management/src/lib/issue-management.component.ts\_* path and observe that the view changes accordingly.

Now, let's have a closer look at some key elements of your project.

#### Main Module

`IssueManagementModule` at the *\_angular/projects/issue-management/src/lib/issue-management.module.ts\_* path is the main module of your module project. There are a few things worth mentioning in it:

- Essential ABP modules, i.e. `CoreModule` and `ThemeSharedModule`, are imported.

- `IssueManagementRoutingModule` is imported.

- `IssueManagementComponent` is declared.

- It is prepared for configurability. The `forLazy` static method enables [a configuration to be passed to the module when it is loaded by the router](https://volosoft.com/blog/how-to-configure-angular-modules-loaded-by-the-router).

#### Main Routing Module

`IssueManagementRoutingModule` at the *\_angular/projects/issue-management/src/lib/issue-management-routing.module.ts\_* path is the main routing module of your module project. It currently does two things:

- Loads `DynamicLayoutComponent` at base path it is given.

- Loads `IssueManagementComponent` as child to the layout, again at the given base path.

You can rearrange this module to load more than one component at different routes, but you need to update the route provider at *\_angular/projects/issue-management/config/src/providers/route.provider.ts\_* to match the new routing structure with the routes in the menu. Please check [Modifying the Menu](../UI/Angular/Modifying-the-Menu.md) to see how route providers work.

#### Config Module

There is a config module at the *\_angular/projects/issue-management/config/src/issue-management-config.module.ts\_* path. The static `forRoot` method of this module is supposed to be called at the route level. So, you may assume the following will take place:

```js

@NgModule({

imports: [

/\* other imports \*/

IssueManagementConfigModule.forRoot(),

],

/\* rest of the module meta data \*/

})

export class AppModule {}

```

You can use this static method to configure an application that uses your module project. An example of such configuration is already implemented and the `ISSUE\_MANAGEMENT\_ROUTE\_PROVIDERS` token is provided here. The method can take options which enables further configuration possibilities.

The difference between the `forRoot` method of the config module and the `forLazy` method of the main module is that, for smallest bundle size, the former should only be used when you have to configure an app before your module is even loaded.

#### Testing Angular UI

Please see the [testing document](../UI/Angular/Testing.md).

## Console

# Console Application Startup Template

This template is used to create a minimalist console application project.

## How to Start With?

First, install the [ABP CLI](../CLI.md) if you haven't installed before:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then use the `abp new` command in an empty folder to create a new solution:

````bash

abp new Acme.MyConsoleApp -t console

````

`Acme.MyConsoleApp` is the solution name, like *\*YourCompany.YourProduct\**. You can use single level, two-levels or three-levels naming.

## Solution Structure

After you use the above command to create a solution, you will have a solution like shown below:

![basic-console-application-solution](../images/basic-console-application-solution.png)

\* `HelloWorldService` is a sample service that implements the `ITransientDependency` interface to register this service to the [dependency injection](../Dependency-Injection.md) system.

## WPF

# WPF Application Startup Template

This template is used to create a minimalist WPF application project.

## How to Start With?

First, install the [ABP CLI](../CLI.md) if you haven't installed before:

````bash

dotnet tool install -g Volo.Abp.Cli

````

Then use the `abp new` command in an empty folder to create a new solution:

````bash

abp new Acme.MyWpfApp -t wpf

````

`Acme.MyWpfApp` is the solution name, like *\*YourCompany.YourProduct\**. You can use single level, two-levels or three-levels naming.

## Solution Structure

After you use the above command to create a solution, you will have a solution like shown below:

![basic-wpf-application-solution](../images/basic-wpf-application-solution.png)

\* `HelloWorldService` is a sample service that implements the `ITransientDependency` interface to register this service to the [dependency injection](../Dependency-Injection.md) system.

# Fundamentals

## Application Startup

## ABP Application Startup

You typically use the [ABP CLI](CLI.md)'s `abp new` command to [get started](Getting-Started.md) with one of the pre-built [startup solution templates](Startup-Templates/Index.md). When you do that, you generally don't need to know the details of how the ABP Framework is integrated with your application or how it is configured and initialized. The startup template also comes with the fundamental ABP packages and [application modules](Modules/Index) are pre-installed and configured for you.

> It is always suggested to [get started with a startup template](Getting-Started.md) and modify it for your requirements. Read this document only if you want to understand the details or if you need to modify how the ABP Framework starts.

While the ABP Framework has a lot of features and integrations, it is built as a lightweight and modular framework. It consists of [hundreds of NuGet and NPM packages](https://abp.io/packages), so you can only use the features you need. If you follow the [Getting Started with an Empty ASP.NET Core MVC / Razor Pages Application](Getting-Started-AspNetCore-Application.md) document, you'll see how easy it is to install the ABP Framework into an empty ASP.NET Core project from scratch. You only need to install a single NuGet package and make a few small changes.

This document is for who wants to better understand how the ABP Framework is initialized and configured on startup.

## Installing to a Console Application

A .NET Console application is the minimalist .NET application. So, it is best to show the installing of the ABP Framework to a console application as a minimalist example.

If you [create a new console application with Visual Studio](https://learn.microsoft.com/en-us/dotnet/core/tutorials/with-visual-studio) (for .NET 7.0 or later), you will see the following solution structure (I named the solution as `MyConsoleDemo`):

![app-startup-console-initial](images/app-startup-console-initial.png)

This example uses the [top level statements](https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/top-level-statements), so it consists of only a single line of code.

The first step is to install the [Volo.Abp.Core](https://www.nuget.org/packages/Volo.Abp.Core) NuGet package, which is the most core NuGet package of the ABP framework. You can install it using the ABP CLI. Execute the following command in the folder of the .csproj file that you want to install the package on:

````bash

abp add-package Volo.Abp.Core

````

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Core).

Alternatively, you can use a command-line terminal in the root folder of the project (the folder containing the `MyConsoleDemo.csproj` file, for this example):

````bash

dotnet add package Volo.Abp.Core

````

After adding the NuGet package, we should create a root [module class](Module-Development-Basics.md) for our application. We can create the following class in the project:

````csharp

using Volo.Abp.Modularity;

namespace MyConsoleDemo

{

public class MyConsoleDemoModule : AbpModule

{

}

}

````

This is an empty class deriving from the `AbpModule` class. It is the main class that you will control your application's dependencies with, and implement your configuration and startup/shutdown logic. For more information, please check the [Modularity](Module-Development-Basics.md) document.

As the second and the last step, change the `Program.cs` as shown in the following code block:

````csharp

using MyConsoleDemo;

using Volo.Abp;

// 1: Create the ABP application container

using var application = await AbpApplicationFactory.CreateAsync<MyConsoleDemoModule>();

// 2: Initialize/start the ABP Framework (and all the modules)

await application.InitializeAsync();

Console.WriteLine("ABP Framework has been started...");

// 3: Stop the ABP Framework (and all the modules)

await application.ShutdownAsync();

````

That's all. Now, ABP Framework is installed, integrated, started and stopped in your application. From now, you can install [ABP packages](https://abp.io/packages) to your application whenever you need them.

## Installing a Framework Package

If you want to send emails from your .NET application, you can use .NET's standard [SmtpClient class](https://learn.microsoft.com/en-us/dotnet/api/system.net.mail.smtpclient). ABP also provides an `IEmailSender` service that simplifies [sending emails](Emailing.md) and configuring the email settings in a central place. If you want to use it, you should install the [Volo.Abp.Emailing](https://www.nuget.org/packages/Volo.Abp.Emailing) NuGet package to your project:

````bash

dotnet add package Volo.Abp.Emailing

````

Once you add a new ABP package/module, you also need to specify the module dependency from your module class. So, change the `MyConsoleDemoModule` class as shown below:

````csharp

using Volo.Abp.Emailing;

using Volo.Abp.Modularity;

namespace MyConsoleDemo

{

[DependsOn(typeof(AbpEmailingModule))] // Added the module dependency

public class MyConsoleDemoModule : AbpModule

{

}

}

````

I've just added a `[DependsOn]` attribute to declare that I want to use the ABP Emailing Module (`AbpEmailingModule`). Now, I can use the `IEmailSender` service in my `Program.cs`:

````csharp

using Microsoft.Extensions.DependencyInjection;

using MyConsoleDemo;

using Volo.Abp;

using Volo.Abp.Emailing;

using var application = await AbpApplicationFactory.CreateAsync<MyConsoleDemoModule>();

await application.InitializeAsync();

// Sending emails using the IEmailSender service

var emailsender = application.ServiceProvider.GetRequiredService<IEmailSender>();

await emailsender.SendAsync(

to: "info@acme.com",

subject: "Hello World",

body: "My message body..."

);

await application.ShutdownAsync();

````

> If you run that application, you get a runtime error indicating that the email sending settings haven't been done yet. You can check the [Email Sending document](Emailing.md) to learn how to configure it.

That's all. Install an ABP NuGet package, add the module dependency (using the `[DependsOn]` attribute) and use any service inside the NuGet package.

The [ABP CLI](CLI.md) already has a special command to perform the addition of an ABP NuGet and also adding the `[DependsOn]` attribute to your module class for you with a single command:

````bash

abp add-package Volo.Abp.Emailing

````

We suggest you to use the `abp add-package` command instead of manually doing it.

## AbpApplicationFactory

`AbpApplicationFactory` is the main class that creates an ABP application container. It provides a single static `CreateAsync` (and `Create` if you can't use asynchronous programming) method with multiple overloads. Let's investigate these overloads to understand where you can use them.

The first overload gets a generic module class parameter as we've used before in this document:

````csharp

AbpApplicationFactory.CreateAsync<MyConsoleDemoModule>();

````

The generic class parameter should be the root module class of your application. All the other modules are resolved as dependencies of that module.

The second overload gets the module class as a `Type` parameter, instead of the generic parameter. So, the previous code block could be re-written as shown below:

````csharp

AbpApplicationFactory.CreateAsync(typeof(MyConsoleDemoModule));

````

Both overloads work exactly the same. So, you can use the second one if you don't know the module class type on development time and you (somehow) calculate it on runtime.

If you create one of the methods above, ABP creates an internal service collection (`IServiceCollection`) and an internal service provider (`IServiceProvider`) to setup the [dependency injection](Dependency-Injection.md) system internally. Notice that we've used the `application.ServiceProvider` property in the *\*Installing a Framework Package\** section to resolve the `IEmailSender` service from the dependency injection system.

The next overload gets an `IServiceCollection` parameter from you to allow you to setup the dependency injection system yourself, or integrate to another framework (like ASP.NET Core) that also sets up the dependency injection system internally.

We can change the `Program.cs` as shown below to externally manage the dependency injection setup:

````csharp

using Microsoft.Extensions.DependencyInjection;

using MyConsoleDemo;

using Volo.Abp;

// 1: Manually created the IServiceCollection

IServiceCollection services = new ServiceCollection();

// 2: Pass the IServiceCollection externally to the ABP Framework

using var application = await AbpApplicationFactory

.CreateAsync<MyConsoleDemoModule>(services);

// 3: Manually built the IServiceProvider object

IServiceProvider serviceProvider = services.BuildServiceProvider();

// 4: Pass the IServiceProvider externally to the ABP Framework

await application.InitializeAsync(serviceProvider);

Console.WriteLine("ABP Framework has been started...");

await application.ShutdownAsync();

````

In this example, we've used .NET's standard dependency injection container. The `services.BuildServiceProvider()` call creates the standard container. However, ABP provides an alternative extension method, `BuildServiceProviderFromFactory()`, that properly works even if you are using another dependency injection container:

````csharp

IServiceProvider serviceProvider = services.BuildServiceProviderFromFactory();

````

> You can check the [Autofac Integration](Autofac-Integration.md) document if you want to learn how you can integrate the [Autofac](https://autofac.org/) dependency injection container with the ABP Framework.

Finally, the `CreateAsync` method has a last overload that takes the module class name as a `Type` parameter and a `IServiceCollection` object. So, we could re-write the last `CreateAsync` method usage as in the following code block:

````csharp

using var application = await AbpApplicationFactory

.CreateAsync(typeof(MyConsoleDemoModule), services);

````

> All of the `CreateAsync` method overloads have `Create` counterparts. If your application type can not utilize asynchronous programming (that means you can't use the `await` keyword), then you can use the `Create` method instead of the `CreateAsync` method.

### AbpApplicationCreationOptions

All of the `CreateAsync` overloads can get an optional `Action<AbpApplicationCreationOptions>` parameter to configure the options that are used on the application creation. See the following example:

````csharp

using var application = await AbpApplicationFactory

.CreateAsync<MyConsoleDemoModule>(options =>

{

options.ApplicationName = "MyApp";

});

````

We've passed a lambda method to configure the `ApplicationName` option. Here's a list of all standard options:

\* `ApplicationName`: A human-readable name for the application. It is a unique value for an application.

\* `Configuration`: Can be used to setup the [application configuration](Configuration.md) when it is not provided by the hosting system. It is not needed for ASP.NET Core and other .NET hosted applications. However, if you've used `AbpApplicationFactory` with an internal service provider, you can use this option to configure how the application configuration is built.

\* `Environment`: Environment name for the application.

\* `PlugInSources`: A list of plugin sources. See the [Plug-In Modules documentation](PlugIn-Modules) to learn how to work with plugins.

\* `Services`: The `IServiceCollection` object that can be used to register service dependencies. You generally don't need that, because you configure your services in your [module class](Module-Development-Basics.md). However, it can be used while writing extension methods for the `AbpApplicationCreationOptions` class.

#### The ApplicationName option

As defined above, the `ApplicationName` option is a human-readable name for the application. It is a unique value for an application.

`ApplicationName` is used by the ABP Framework in several places to distinguish the application. For example, the [audit logging](Audit-Logging.md) system saves the `ApplicationName` in each audit log record written by the related application, so you can understand which application has created the audit log entry. So, if your system consists of multiple applications (like a microservice solution) that are saving audit logs to a single point, you should be sure that each application has a different `ApplicationName`.

The `ApplicationName` property's value is set automatically from the **\*\*entry assembly's name\*\*** (generally, the project name in a .NET solution) by default, which is proper for most cases, since each application typically has a unique entry assembly name.

There are two ways to set the application name to a different value. In this first approach, you can set the `ApplicationName` property in your application's [configuration](Configuration.md). The easiest way is to add an `ApplicationName` field to your `appsettings.json` file:

````json

{

"ApplicationName": "Services.Ordering"

}

````

Alternatively, you can set `AbpApplicationCreationOptions.ApplicationName` while creating the ABP application. You can find the `AddApplication` or `AddApplicationAsync` call in your solution (typically in the `Program.cs` file), and set the `ApplicationName` option as shown below:

````csharp

await builder.AddApplicationAsync<OrderingServiceHttpApiHostModule>(options =>

{

options.ApplicationName = "Services.Ordering";

});

````

#### IApplicationInfoAccessor

If you need to access the `ApplicationName` later in your solution, you can inject the `IApplicationInfoAccessor` service and get the value from its `ApplicationName` property.

`IApplicationInfoAccessor` also provides an `InstanceId` value, that is a random GUID value that is generated when your application starts. You can use that value to distinguish application instances from each other.

## IAbpApplication

`AbpApplicationFactory` returns an `IAbpApplication` object from its `CreateAsync` (or `Create`) method. `IAbpApplication` is the main container for an ABP application. It is also registered to the [dependency injection](Dependency-Injection.md) system, so you can inject `IAbpApplication` in your services to use its properties and methods.

Here's a list of `IAbpApplication` properties you may want to know:

\* `StartupModuleType`: Gets the root module of the application that was used while creating the application container (on the `AbpApplicationFactory.CreateAsync` method).

\* `Services`: A list of all service registrations (the `IServiceCollection` object). You can not add new services to this collection after application initialization (you can actually add, but it won't have any effect).

\* `ServiceProvider`: A reference to the root service provider used by the application. This can not be used before initializing the application. If you need to resolve non-singleton services from that `IServiceProvider` object, always create a new service scope and dispose it after usage. Otherwise, your application will have memory leak problems. See the *\*Releasing/Disposing Services\** section of the [dependency injection](Dependency-Injection.md) document for more information about service scopes.

\* `Modules`: A read-only list of all the modules loaded into the current application. Alternatively, you can inject the `IModuleContainer` service if you need to access the module list in your application code.

The `IAbpApplication` interface extends the `IApplicationInfoAccessor` interface, so you can get the `ApplicationName` and `InstanceId` values from it. However, if you only need to access these properties, inject and use the `IApplicationInfoAccessor` service instead.

`IAbpApplication` is disposable. Always dispose of it before exiting your application.

## IAbpHostEnvironment

Sometimes, while creating an application, we need to get the current hosting environment and take actions according to that. In such cases, we can use some services such as [IWebHostEnvironment](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.iwebhostenvironment?view=aspnetcore-7.0) or [IWebAssemblyHostEnvironment](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.webassembly.hosting.iwebassemblyhostenvironment) provided by .NET, in the final application.

However, we can not use these services in a class library, which is used by the final application. ABP Framework provides the `IAbpHostEnvironment` service, which allows you to get the current environment name whenever you want. `IAbpHostEnvironment` is used by the ABP Framework in several places to perform specific actions by the environment. For example, ABP Framework reduces the cache duration on the **\*\*Development\*\*** environment for some services.

`IAbpHostEnvironment` obtains the current environment name by the following order:

1. Gets and sets the environment name if it's specified in the `AbpApplicationCreationOptions`.

2. Tries to obtain the environment name from the `IWebHostEnvironment` or `IWebAssemblyHostEnvironment` services for ASP.NET Core & Blazor WASM applications if the environment name isn't specified in the `AbpApplicationCreationOptions`.

3. Sets the environment name as **\*\*Production\*\***, if the environment name is not specified or can not be obtained from the services.

You can configure the `AbpApplicationCreationOptions` [options class](Options.md) while creating the ABP application and set an environment name to its `Environment` property. You can find the `AddApplication` or `AddApplicationAsync` call in your solution (typically in the `Program.cs` file), and set the `Environment` option as shown below:

```csharp

await builder.AddApplicationAsync<OrderingServiceHttpApiHostModule>(options =>

{

options.Environment = Environments.Staging; //or directly set as "Staging"

});

```

Then, whenever you need to get the current environment name or check the environment, you can use the `IAbpHostEnvironment` interface:

```csharp

public class MyDemoService

{

private readonly IAbpHostEnvironment \_abpHostEnvironment;

public MyDemoService(IAbpHostEnvironment abpHostEnvironment)

{

\_abpHostEnvironment = abpHostEnvironment;

}

public void MyMethod()

{

var environmentName = \_abpHostEnvironment.EnvironmentName;

if (\_abpHostEnvironment.IsDevelopment()) { /\* ... \*/ }

if (\_abpHostEnvironment.IsStaging()) { /\* ... \*/ }

if (\_abpHostEnvironment.IsProduction()) { /\* ... \*/ }

if (\_abpHostEnvironment.IsEnvironment("custom-environment")) { /\* ... \*/ }

}

}

```

## .NET Generic Host & ASP.NET Core Integrations

`AbpApplicationFactory` can create a standalone ABP application container without any external dependency. However, in most cases, you will want to integrate it with [.NET's generic host](https://learn.microsoft.com/en-us/dotnet/core/extensions/generic-host) or ASP.NET Core. For such usages, ABP provides built-in extension methods to easily create an ABP application container that is well-integrated to these systems.

The [Getting Started with an Empty ASP.NET Core MVC / Razor Pages Application](Getting-Started-AspNetCore-Application.md) document clearly explains how you can create an ABP application container in an ASP.NET Core application.

You can also [create a console application](Startup-Templates/Console) to see how it is integrated with .NET Generic Host.

> Most of the times, you will directly create ABP applications using the ABP CLI's `new` command. So, you don't need to care about these integration details.

## See Also

\* [Dependency injection](Dependency-Injection.md)

\* [Modularity](Module-Development-Basics.md)

## Authorization

# Authorization

Authorization is used to check if a user is allowed to perform some specific operations in the application.

ABP extends [ASP.NET Core Authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction) by adding **\*\*permissions\*\*** as auto [policies](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies) and allowing authorization system to be usable in the **\*\*[**application services**](Application-Services.md)\*\*** too.

So, all the ASP.NET Core authorization features and the documentation are valid in an ABP based application. This document focuses on the features that are added on top of ASP.NET Core authorization features.

## Authorize Attribute

ASP.NET Core defines the [\*\*Authorize\*\*](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/simple) attribute that can be used for an action, a controller or a page. ABP allows you to use the same attribute for an [application service](Application-Services.md) too.

Example:

```csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Services;

namespace Acme.BookStore

{

[Authorize]

public class AuthorAppService : ApplicationService, IAuthorAppService

{

public Task<List<AuthorDto>> GetListAsync()

{

...

}

[AllowAnonymous]

public Task<AuthorDto> GetAsync(Guid id)

{

...

}

[Authorize("BookStore\_Author\_Create")]

public Task CreateAsync(CreateAuthorDto input)

{

...

}

}

}

```

- `Authorize` attribute forces the user to login into the application in order to use the `AuthorAppService` methods. So, `GetListAsync` method is only available to the authenticated users.

- `AllowAnonymous` suppresses the authentication. So, `GetAsync` method is available to everyone including unauthorized users.

- `[Authorize("BookStore\_Author\_Create")]` defines a policy (see [policy based authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies)) that is checked to authorize the current user.

"BookStore\_Author\_Create" is an arbitrary policy name. If you declare an attribute like that, ASP.NET Core authorization system expects a policy to be defined before.

You can, of course, implement your policies as stated in the ASP.NET Core documentation. But for simple true/false conditions like a policy was granted to a user or not, ABP defines the permission system which will be explained in the next section.

## Permission System

A permission is a simple policy that is granted or prohibited for a particular user, role or client.

### Defining Permissions

To define permissions, create a class inheriting from the `PermissionDefinitionProvider` as shown below:

```csharp

using Volo.Abp.Authorization.Permissions;

namespace Acme.BookStore.Permissions

{

public class BookStorePermissionDefinitionProvider : PermissionDefinitionProvider

{

public override void Define(IPermissionDefinitionContext context)

{

var myGroup = context.AddGroup("BookStore");

myGroup.AddPermission("BookStore\_Author\_Create");

}

}

}

```

> ABP automatically discovers this class. No additional configuration required!

> You typically define this class inside the `Application.Contracts` project of your [application](Startup-Templates/Application.md). The startup template already comes with an empty class named *\*YourProjectNamePermissionDefinitionProvider\** that you can start with.

In the `Define` method, you first need to add a **\*\*permission group\*\*** or get an existing group then add **\*\*permissions\*\*** to this group.

When you define a permission, it becomes usable in the ASP.NET Core authorization system as a **\*\*policy\*\*** name. It also becomes visible in the UI. See permissions dialog for a role:

![authorization-new-permission-ui](images/authorization-new-permission-ui.png)

- The "BookStore" group is shown as a new tab on the left side.

- "BookStore\_Author\_Create" on the right side is the permission name. You can grant or prohibit it for the role.

When you save the dialog, it is saved to the database and used in the authorization system.

> The screen above is available when you have installed the identity module, which is basically used for user and role management. Startup templates come with the identity module pre-installed.

#### Localizing the Permission Name

"BookStore\_Author\_Create" is not a good permission name for the UI. Fortunately, `AddPermission` and `AddGroup` methods can take `LocalizableString` as second parameters:

```csharp

var myGroup = context.AddGroup(

"BookStore",

LocalizableString.Create<BookStoreResource>("BookStore")

);

myGroup.AddPermission(

"BookStore\_Author\_Create",

LocalizableString.Create<BookStoreResource>("Permission:BookStore\_Author\_Create")

);

```

Then you can define texts for "BookStore" and "Permission:BookStore\_Author\_Create" keys in the localization file:

```json

"BookStore": "Book Store",

"Permission:BookStore\_Author\_Create": "Creating a new author"

```

> For more information, see the [localization document](Localization.md) on the localization system.

The localized UI will be as seen below:

![authorization-new-permission-ui-localized](images/authorization-new-permission-ui-localized.png)

#### Multi-Tenancy

ABP supports [multi-tenancy](Multi-Tenancy.md) as a first class citizen. You can define multi-tenancy side option while defining a new permission. It gets one of the three values defined below:

- **\*\*Host\*\***: The permission is available only for the host side.

- **\*\*Tenant\*\***: The permission is available only for the tenant side.

- **\*\*Both\*\*** (default): The permission is available both for tenant and host sides.

> If your application is not multi-tenant, you can ignore this option.

To set the multi-tenancy side option, pass to the third parameter of the `AddPermission` method:

```csharp

myGroup.AddPermission(

"BookStore\_Author\_Create",

LocalizableString.Create<BookStoreResource>("Permission:BookStore\_Author\_Create"),

multiTenancySide: MultiTenancySides.Tenant //set multi-tenancy side!

);

```

#### Enable/Disable Permissions

A permission is enabled by default. It is possible to disable a permission. A disabled permission will be prohibited for everyone. You can still check for the permission, but it will always return prohibited.

Example definition:

````csharp

myGroup.AddPermission("Author\_Management", isEnabled: false);

````

You normally don't need to define a disabled permission (unless you temporary want disable a feature of your application). However, you may want to disable a permission defined in a depended module. In this way you can disable the related application functionality. See the "*\*Changing Permission Definitions of a Depended Module\**" section below for an example usage.

> Note: Checking an undefined permission will throw an exception while a disabled permission check simply returns prohibited (false).

#### Child Permissions

A permission may have child permissions. It is especially useful when you want to create a hierarchical permission tree where a permission may have additional sub permissions which are available only if the parent permission has been granted.

Example definition:

```csharp

var authorManagement = myGroup.AddPermission("Author\_Management");

authorManagement.AddChild("Author\_Management\_Create\_Books");

authorManagement.AddChild("Author\_Management\_Edit\_Books");

authorManagement.AddChild("Author\_Management\_Delete\_Books");

```

The result on the UI is shown below (you probably want to localize permissions for your application):

![authorization-new-permission-ui-hierarcy](images/authorization-new-permission-ui-hierarcy.png)

For the example code, it is assumed that a role/user with "Author\_Management" permission granted may have additional permissions. Then a typical application service that checks permissions can be defined as shown below:

```csharp

[Authorize("Author\_Management")]

public class AuthorAppService : ApplicationService, IAuthorAppService

{

public Task<List<AuthorDto>> GetListAsync()

{

...

}

public Task<AuthorDto> GetAsync(Guid id)

{

...

}

[Authorize("Author\_Management\_Create\_Books")]

public Task CreateAsync(CreateAuthorDto input)

{

...

}

[Authorize("Author\_Management\_Edit\_Books")]

public Task UpdateAsync(CreateAuthorDto input)

{

...

}

[Authorize("Author\_Management\_Delete\_Books")]

public Task DeleteAsync(CreateAuthorDto input)

{

...

}

}

```

- `GetListAsync` and `GetAsync` will be available to users if they have `Author\_Management` permission is granted.

- Other methods require additional permissions.

### Overriding a Permission by a Custom Policy

If you define and register a policy to the ASP.NET Core authorization system with the same name of a permission, your policy will override the existing permission. This is a powerful way to extend the authorization for a pre-built module that you are using in your application.

See [policy based authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies) document to learn how to define a custom policy.

### Changing Permission Definitions of a Depended Module

A class deriving from the `PermissionDefinitionProvider` (just like the example above) can also get existing permission definitions (defined by the depended [modules](Module-Development-Basics.md)) and change their definitions.

Example:

````csharp

context

.GetPermissionOrNull(IdentityPermissions.Roles.Delete)

.IsEnabled = false;

````

When you write this code inside your permission definition provider, it finds the "role deletion" permission of the [Identity Module](Modules/Identity.md) and disabled the permission, so no one can delete a role on the application.

> Tip: It is better to check the value returned by the `GetPermissionOrNull` method since it may return null if the given permission was not defined.

### Permission Depending on a Condition

You may want to disable a permission based on a condition. Disabled permissions are not visible on the UI and always returns `prohibited` when you check them. There are two built-in conditional dependencies for a permission definition;

\* A permission can be automatically disabled if a [Feature](Features.md) was disabled.

\* A permission can be automatically disabled if a [Global Feature](Global-Features.md) was disabled.

In addition, you can create your custom extensions.

#### Depending on a Features

Use the `RequireFeatures` extension method on your permission definition to make the permission available only if a given feature is enabled:

````csharp

myGroup.AddPermission("Book\_Creation")

.RequireFeatures("BookManagement");

````

#### Depending on a Global Feature

Use the `RequireGlobalFeatures` extension method on your permission definition to make the permission available only if a given feature is enabled:

````csharp

myGroup.AddPermission("Book\_Creation")

.RequireGlobalFeatures("BookManagement");

````

#### Creating a Custom Permission Dependency

`PermissionDefinition` supports state check, Please refer to [Simple State Checker's documentation](SimpleStateChecker.md)

## IAuthorizationService

ASP.NET Core provides the `IAuthorizationService` that can be used to check for authorization. Once you inject, you can use it in your code to conditionally control the authorization.

Example:

```csharp

public async Task CreateAsync(CreateAuthorDto input)

{

var result = await AuthorizationService

.AuthorizeAsync("Author\_Management\_Create\_Books");

if (result.Succeeded == false)

{

//throw exception

throw new AbpAuthorizationException("...");

}

//continue to the normal flow...

}

```

> `AuthorizationService` is available as a property when you derive from ABP's `ApplicationService` base class. Since it is widely used in application services, `ApplicationService` pre-injects it for you. Otherwise, you can directly [inject](Dependency-Injection.md) it into your class.

Since this is a typical code block, ABP provides extension methods to simplify it.

Example:

```csharp

public async Task CreateAsync(CreateAuthorDto input)

{

await AuthorizationService.CheckAsync("Author\_Management\_Create\_Books");

//continue to the normal flow...

}

```

`CheckAsync` extension method throws `AbpAuthorizationException` if the current user/client is not granted for the given permission. There is also `IsGrantedAsync` extension method that returns `true` or `false`.

`IAuthorizationService` has some overloads for the `AuthorizeAsync` method. These are explained in the [ASP.NET Core authorization documentation](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction).

> Tip: Prefer to use the `Authorize` attribute wherever possible, since it is declarative & simple. Use `IAuthorizationService` if you need to conditionally check a permission and run a business code based on the permission check.

## Check a Permission in JavaScript

See the following documents to learn how to re-use the authorization system on the client side:

\* [ASP.NET Core MVC / Razor Pages UI: Authorization](UI/AspNetCore/JavaScript-API/Auth.md)

\* [Angular UI Authorization](UI/Angular/Permission-Management.md)

\* [Blazor UI Authorization](UI/Blazor/Authorization.md)

## Permission Management

Permission management is normally done by an admin user using the permission management modal:

![authorization-new-permission-ui-localized](images/authorization-new-permission-ui-localized.png)

If you need to manage permissions by code, inject the `IPermissionManager` and use as shown below:

```csharp

public class MyService : ITransientDependency

{

private readonly IPermissionManager \_permissionManager;

public MyService(IPermissionManager permissionManager)

{

\_permissionManager = permissionManager;

}

public async Task GrantPermissionForUserAsync(Guid userId, string permissionName)

{

await \_permissionManager.SetForUserAsync(userId, permissionName, true);

}

public async Task ProhibitPermissionForUserAsync(Guid userId, string permissionName)

{

await \_permissionManager.SetForUserAsync(userId, permissionName, false);

}

}

```

`SetForUserAsync` sets the value (true/false) for a permission of a user. There are more extension methods like `SetForRoleAsync` and `SetForClientAsync`.

`IPermissionManager` is defined by the permission management module. See the [permission management module documentation](Modules/Permission-Management.md) for more information.

## Advanced Topics

### Permission Value Providers

Permission checking system is extensible. Any class derived from `PermissionValueProvider` (or implements `IPermissionValueProvider`) can contribute to the permission check. There are three pre-defined value providers:

- `UserPermissionValueProvider` checks if the current user is granted for the given permission. It gets user id from the current claims. User claim name is defined with the `AbpClaimTypes.UserId` static property.

- `RolePermissionValueProvider` checks if any of the roles of the current user is granted for the given permission. It gets role names from the current claims. Role claims name is defined with the `AbpClaimTypes.Role` static property.

- `ClientPermissionValueProvider` checks if the current client is granted for the given permission. This is especially useful on a machine to machine interaction where there is no current user. It gets the client id from the current claims. Client claim name is defined with the `AbpClaimTypes.ClientId` static property.

You can extend the permission checking system by defining your own permission value provider.

Example:

```csharp

public class SystemAdminPermissionValueProvider : PermissionValueProvider

{

public SystemAdminPermissionValueProvider(IPermissionStore permissionStore)

: base(permissionStore)

{

}

public override string Name => "SystemAdmin";

public async override Task<PermissionGrantResult>

CheckAsync(PermissionValueCheckContext context)

{

if (context.Principal?.FindFirst("User\_Type")?.Value == "SystemAdmin")

{

return PermissionGrantResult.Granted;

}

return PermissionGrantResult.Undefined;

}

}

```

This provider allows for all permissions to a user with a `User\_Type` claim that has `SystemAdmin` value. It is common to use current claims and `IPermissionStore` in a permission value provider.

A permission value provider should return one of the following values from the `CheckAsync` method:

- `PermissionGrantResult.Granted` is returned to grant the user for the permission. If any of the providers return `Granted`, the result will be `Granted`, if no other provider returns `Prohibited`.

- `PermissionGrantResult.Prohibited` is returned to prohibit the user for the permission. If any of the providers return `Prohibited`, the result will always be `Prohibited`. Doesn't matter what other providers return.

- `PermissionGrantResult.Undefined` is returned if this value provider could not decide about the permission value. Return this to let other providers check the permission.

Once a provider is defined, it should be added to the `AbpPermissionOptions` as shown below:

```csharp

Configure<AbpPermissionOptions>(options =>

{

options.ValueProviders.Add<SystemAdminPermissionValueProvider>();

});

```

### Permission Store

`IPermissionStore` is the only interface that needs to be implemented to read the value of permissions from a persistence source, generally a database system. The Permission Management module implements it and pre-installed in the application startup template. See the [permission management module documentation](Modules/Permission-Management.md) for more information

### AlwaysAllowAuthorizationService

`AlwaysAllowAuthorizationService` is a class that is used to bypass the authorization service. It is generally used in integration tests where you may want to disable the authorization system.

Use `IServiceCollection.AddAlwaysAllowAuthorization()` extension method to register the `AlwaysAllowAuthorizationService` to the [dependency injection](Dependency-Injection.md) system:

```csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAlwaysAllowAuthorization();

}

```

This is already done for the startup template integration tests.

### Claims Principal Factory

Claims are important elements of authentication and authorization. ABP uses the `IAbpClaimsPrincipalFactory` service to create claims on authentication. This service was designed as extensible. If you need to add your custom claims to the authentication ticket, you can implement the `IAbpClaimsPrincipalContributor` in your application.

**\*\*Example: Add a** `SocialSecurityNumber` **claim and get it:\*\***

```csharp

public class SocialSecurityNumberClaimsPrincipalContributor : IAbpClaimsPrincipalContributor, ITransientDependency

{

public async Task ContributeAsync(AbpClaimsPrincipalContributorContext context)

{

var identity = context.ClaimsPrincipal.Identities.FirstOrDefault();

var userId = identity?.FindUserId();

if (userId.HasValue)

{

var userService = context.ServiceProvider.GetRequiredService<IUserService>(); //Your custom service

var socialSecurityNumber = await userService.GetSocialSecurityNumberAsync(userId.Value);

if (socialSecurityNumber != null)

{

identity.AddClaim(new Claim("SocialSecurityNumber", socialSecurityNumber));

}

}

}

}

public static class CurrentUserExtensions

{

public static string GetSocialSecurityNumber(this ICurrentUser currentUser)

{

return currentUser.FindClaimValue("SocialSecurityNumber");

}

}

```

> If you use Identity Server please add your claims to `RequestedClaims` of `AbpClaimsServiceOptions`.

```csharp

Configure<AbpClaimsServiceOptions>(options =>

{

options.RequestedClaims.AddRange(new[]{ "SocialSecurityNumber" });

});

```

## See Also

\* [Permission Management Module](Modules/Permission-Management.md)

\* [ASP.NET Core MVC / Razor Pages JavaScript Auth API](UI/AspNetCore/JavaScript-API/Auth.md)

\* [Permission Management in Angular UI](UI/Angular/Permission-Management.md)

## Caching

# Distributed Caching

ABP Framework extends the [ASP.NET Core distributed cache](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed).

> **\*\*Default implementation of the** `IDistributedCache` **interface is**` MemoryDistributedCache` **which works in-memory.\*\*** See [ASP.NET Core's documentation](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed) to see how to switch to Redis or another cache provider. Also, see the [Redis Cache](Redis-Cache.md) document if you want to use Redis as the distributed cache server.

## Installation

> This package is already installed by default with the [application startup template](Startup-Templates/Application.md). So, most of the time, you don't need to install it manually.

[Volo.Abp.Caching](https://www.nuget.org/packages/Volo.Abp.Caching) is the main package of the caching system. You can install it a project using the add-package command of the [ABP CLI](CLI.md):

```bash

abp add-package Volo.Abp.Caching

```

You need to run this command on a command line terminal in a folder containing a `csproj` file (see [other options](https://abp.io/package-detail/Volo.Abp.Caching) to install).

## Usage

### `IDistributedCache` Interface

ASP.NET Core defines the `IDistributedCache` interface to get/set the cache values. But it has some difficulties:

\* It works with **\*\*byte arrays\*\*** rather than .NET objects. So, you need to **\*\*serialize/deserialize\*\*** the objects you need to cache.

\* It provides a **\*\*single key pool\*\*** for all cache items, so;

\* You need to care about the keys to distinguish **\*\*different type of objects\*\***.

\* You need to care about the cache items of **\*\*different tenants\*\*** in a [multi-tenant](Multi-Tenancy.md) system.

> `IDistributedCache` is defined in the `Microsoft.Extensions.Caching.Abstractions` package. That means it is not only usable for ASP.NET Core applications, but also available to **\*\*any type of applications\*\***.

See [ASP.NET Core's distributed caching document](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed) for more information.

### `IDistributedCache<TCacheItem>` Interface

ABP framework defines the generic `IDistributedCache<TCacheItem>` interface in the [Volo.Abp.Caching](https://www.nuget.org/packages/Volo.Abp.Caching/) package. `TCacheItem` is the type of the object stored in the cache.

`IDistributedCache<TCacheItem>` solves the difficulties explained above;

\* It internally **\*\*serializes/deserializes\*\*** the cached objects. Uses **\*\*JSON\*\*** serialization by default, but can be overridden by replacing the `IDistributedCacheSerializer` service in the [dependency injection](Dependency-Injection.md) system.

\* It automatically adds a **\*\*cache name\*\*** prefix to the cache keys based on the object type stored in the cache. Default cache name is the full name of the cache item class (`CacheItem` postfix is removed if your cache item class ends with it). You can use the **\*\***`CacheName` **attribute\*\*** on the cache item class to set the cache name.

\* It automatically adds the **\*\*current tenant id\*\*** to the cache key to distinguish cache items for different tenants (if your application is [multi-tenant](Multi-Tenancy.md)). Define `IgnoreMultiTenancy` attribute on the cache item class to disable this if you want to share the cached objects among all tenants in a multi-tenant application.

\* Allows to define a **\*\*global cache key prefix\*\*** per application, so different applications can use their isolated key pools in a shared distributed cache server.

\* It **\*\*can tolerate errors\*\*** wherever possible and bypasses the cache. This is useful when you have temporary problems on the cache server.

\* It has methods like `GetManyAsync` and `SetManyAsync` which significantly improve the performance on **\*\*batch operations\*\***.

**\*\*Example: Store Book names and prices in the cache\*\***

````csharp

namespace MyProject

{

public class BookCacheItem

{

public string Name { get; set; }

public float Price { get; set; }

}

}

````

You can inject and use the `IDistributedCache<BookCacheItem>` service to get/set `BookCacheItem` objects:

````csharp

using System;

using System.Threading.Tasks;

using Microsoft.Extensions.Caching.Distributed;

using Volo.Abp.Caching;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class BookService : ITransientDependency

{

private readonly IDistributedCache<BookCacheItem> \_cache;

public BookService(IDistributedCache<BookCacheItem> cache)

{

\_cache = cache;

}

public async Task<BookCacheItem> GetAsync(Guid bookId)

{

return await \_cache.GetOrAddAsync(

bookId.ToString(), //Cache key

async () => await GetBookFromDatabaseAsync(bookId),

() => new DistributedCacheEntryOptions

{

AbsoluteExpiration = DateTimeOffset.Now.AddHours(1)

}

);

}

private Task<BookCacheItem> GetBookFromDatabaseAsync(Guid bookId)

{

//TODO: get from database

}

}

}

````

\* This sample service uses the `GetOrAddAsync()` method to get a book item from the cache. `GetOrAddAsync` is an additional method that was added by the ABP Framework to the standard ASP.NET Core distributed cache methods.

\* If the book was not found in the cache, it calls the factory method (`GetBookFromDatabaseAsync` in this case) to retrieve the book item from the original source.

\* `GetOrAddAsync` optionally gets a `DistributedCacheEntryOptions` which can be used to set the lifetime of the cached item.

`IDistributedCache<BookCacheItem>` supports the same methods of the ASP.NET Core's standard `IDistributedCache` interface, so you can refer [it's documentation](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed).

### `IDistributedCache<TCacheItem, TCacheKey>` Interface

`IDistributedCache<TCacheItem>` interface assumes that the type of your **\*\*cache key\*\*** is `string` (so, you need to manually convert your key to string if you need to use a different kind of cache key). While this is not a big deal, `IDistributedCache<TCacheItem, TCacheKey>` can be used when your cache key type is not `string`.

**\*\*Example: Store Book names and prices in the cache\*\***

````csharp

using Volo.Abp.Caching;

namespace MyProject

{

[CacheName("Books")]

public class BookCacheItem

{

public string Name { get; set; }

public float Price { get; set; }

}

}

````

\* This example uses the `CacheName` attribute for the `BookCacheItem` class to set the cache name.

You can inject and use the `IDistributedCache<BookCacheItem, Guid>` service to get/set `BookCacheItem` objects:

````csharp

using System;

using System.Threading.Tasks;

using Microsoft.Extensions.Caching.Distributed;

using Volo.Abp.Caching;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class BookService : ITransientDependency

{

private readonly IDistributedCache<BookCacheItem, Guid> \_cache;

public BookService(IDistributedCache<BookCacheItem, Guid> cache)

{

\_cache = cache;

}

public async Task<BookCacheItem> GetAsync(Guid bookId)

{

return await \_cache.GetOrAddAsync(

bookId, //Guid type used as the cache key

async () => await GetBookFromDatabaseAsync(bookId),

() => new DistributedCacheEntryOptions

{

AbsoluteExpiration = DateTimeOffset.Now.AddHours(1)

}

);

}

private Task<BookCacheItem> GetBookFromDatabaseAsync(Guid bookId)

{

//TODO: get from database

}

}

}

````

\* This sample service uses the `GetOrAddAsync()` method to get a book item from the cache.

\* Since cache explicitly implemented as using `Guid` as cache key, `Guid` value passed to `\_cache\_GetOrAddAsync()` method.

#### Complex Types as the Cache Key

`IDistributedCache<TCacheItem, TCacheKey>` internally uses `ToString()` method of the key object to convert it to a string. If you need to use a complex object as the cache key, you need to override `ToString` method of your class.

An example class that is used as a cache key:

````csharp

public class UserInOrganizationCacheKey

{

public Guid UserId { get; set; }

public Guid OrganizationId { get; set; }

//Builds the cache key

public override string ToString()

{

return $"{UserId}\_{OrganizationId}";

}

}

````

Example usage:

````csharp

public class BookService : ITransientDependency

{

private readonly IDistributedCache<UserCacheItem, UserInOrganizationCacheKey> \_cache;

public BookService(

IDistributedCache<UserCacheItem, UserInOrganizationCacheKey> cache)

{

\_cache = cache;

}

...

}

````

## Configuration

### AbpDistributedCacheOptions

`AbpDistributedCacheOptions` is the main [options class](Options.md) to configure the caching.

**\*\*Example: Set the cache key prefix for the application\*\***

```csharp

Configure<AbpDistributedCacheOptions>(options =>

{

options.KeyPrefix = "MyApp1";

});

```

> Write that code inside the `ConfigureServices` method of your [module class](Module-Development-Basics.md).

#### Available Options

\* `HideErrors` (`bool`, default: `true`): Enables/disables hiding the errors on writing/reading values from the cache server.

\* `KeyPrefix` (`string`, default: `null`): If your cache server is shared by multiple applications, you can set a prefix for the cache keys for your application. In this case, different applications can not overwrite each other's cache items.

\* `GlobalCacheEntryOptions` (`DistributedCacheEntryOptions`): Used to set default distributed cache options (like `AbsoluteExpiration` and `SlidingExpiration`) used when you don't specify the options while saving cache items. Default value uses the `SlidingExpiration` as 20 minutes.

## Error Handling

When you design a cache for your objects, you typically try to get the value from cache first. If not found in the cache, you query the object from the **\*\*original source\*\***. It may be located in a **\*\*database\*\*** or may require to perform an HTTP call to a remote server.

In most cases, you want to **\*\*tolerate the cache errors\*\***; If you get error from the cache server you don't want to cancel the operation. Instead, you silently hide (and log) the error and **\*\*query from the original source\*\***. This is what the ABP Framework does by default.

ABP's Distributed Cache [handle](Exception-Handling.md), log and hide errors by default. There is an option to change this globally (see the options below).

In addition, all of the `IDistributedCache<TCacheItem>` (and `IDistributedCache<TCacheItem, TCacheKey>`) methods have an optional `hideErrors` parameter, which is `null` by default. The global value is used if this parameter left as `null`, otherwise you can decide to hide or throw the exceptions for individual method calls.

## Batch Operations

ABP's distributed cache interfaces provide methods to perform batch methods those improves the performance when you want to batch operation multiple cache items in a single method call.

\* `SetManyAsync` and `SetMany` methods can be used to set multiple values to the cache.

\* `GetManyAsync` and `GetMany` methods can be used to retrieve multiple values from the cache.

\* `GetOrAddManyAsync` and `GetOrAddMany` methods can be used to retrieve multiple values and set missing values from the cache

\* `RefreshManyAsync` and `RefreshMany` methods can be used to resets the sliding expiration timeout of multiple values from the cache

\* `RemoveManyAsync` and `RemoveMany` methods can be used to remove multiple values from the cache

> These are not standard methods of the ASP.NET Core caching. So, some providers may not support them. They are supported by the [ABP Redis Cache integration package](Redis-Cache.md). If the provider doesn't support, it fallbacks to `SetAsync` and `GetAsync` ... methods (called once for each item).

## Caching Entities

ABP Framework provides a [Distributed Entity Cache System](Entity-Cache.md) for caching entities. It is useful if you want to use caching for quicker access to the entity rather than repeatedly querying it from the database.

It's designed as read-only and automatically invalidates a cached entity if the entity is updated or deleted.

> See the [Entity Cache](Entity-Cache.md) documentation for more information.

## Advanced Topics

### Unit Of Work Level Cache

Distributed cache service provides an interesting feature. Assume that you've updated the price of a book in the database, then set the new price to the cache, so you can use the cached value later. What if you have an exception after setting the cache and you **\*\*rollback the transaction\*\*** that updates the price of the book? In this case, cache value will be incorrect.

`IDistributedCache<..>` methods gets an optional parameter, named `considerUow`, which is `false` by default. If you set it to `true`, then the changes you made for the cache are not actually applied to the real cache store, but associated with the current [unit of work](Unit-Of-Work.md). You get the value you set in the same unit of work, but the changes are applied **\*\*only if the current unit of work succeed\*\***.

### IDistributedCacheSerializer

`IDistributedCacheSerializer` service is used to serialize and deserialize the cache items. Default implementation is the `Utf8JsonDistributedCacheSerializer` class that uses `IJsonSerializer` service to convert objects to [JSON](Json-Serialization.md) and vice verse. Then it uses UTC8 encoding to convert the JSON string to a byte array which is accepted by the distributed cache.

You can [replace](Dependency-Injection.md) this service by your own implementation if you want to implement your own serialization logic.

### IDistributedCacheKeyNormalizer

`IDistributedCacheKeyNormalizer` is implemented by the `DistributedCacheKeyNormalizer` class by default. It adds cache name, application cache prefix and current tenant id to the cache key. If you need a more advanced key normalization, you can [replace](Dependency-Injection.md) this service by your own implementation.

## See Also

\* [Entity Cache](Entity-Cache.md)

\* [Redis Cache](Redis-Cache.md)

### Entity Cache

# Entity Cache

ABP Framework provides an entity caching system that works on top of the [distributed caching](Caching.md) system. It does the following operations on behalf of you:

\* Gets the entity from the database (by using the [repositories](Repositories.md)) in its first call and then gets it from the cache in subsequent calls.

\* Automatically invalidates the cached entity if the entity is updated or deleted. Thus, it will be retrieved from the database in the next call and will be re-cached.

## Caching Entity Objects

`IEntityCache<TEntityCacheItem, TKey>` is a simple service provided by the ABP Framework for caching entities. Assume that you have a `Product` entity as shown below:

```csharp

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

public string Description { get; set; }

public float Price { get; set; }

public int StockCount { get; set; }

}

```

If you want to cache this entity, you should first configure the [dependency injection](Dependency-Injection.md) system to register the `IEntityCache` service in the `ConfigureServices` method of your [module class](Module-Development-Basics.md):

```csharp

context.Services.AddEntityCache<Product, Guid>();

```

Now you can inject the `IEntityCache<Product, Guid>` service wherever you need:

```csharp

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IEntityCache<Product, Guid> \_productCache;

public ProductAppService(IEntityCache<Product, Guid> productCache)

{

\_productCache = productCache;

}

public async Task<ProductDto> GetAsync(Guid id)

{

var product = await \_productCache.GetAsync(id);

return ObjectMapper.Map<Product, ProductDto>(product);

}

}

```

> Note that we've used the `ObjectMapper` service to map from `Product` to `ProductDto`. You should configure that [object mapping](Object-To-Object-Mapping.md) to make that example service properly works.

That's all. The cache name (in the distributed cache server) will be full name (with namespace) of the `Product` class. You can use the `[CacheName]` attribute to change it. Please refer to the [caching document](Caching.md) for details.

## Using a Cache Item Class

In the previous section, we've directly cached the `Product` entity. In that case, the `Product` class must be serializable to JSON (and deserializable from JSON). Sometimes that might not be possible or you may want to use another class to store the cache data. For example, we may want to use the `ProductDto` class instead of the `Product` class for the cached object if the `Product` entity.

Assume that we've created a `ProductDto` class as shown below:

```csharp

public class ProductDto : EntityDto<Guid>

{

public string Name { get; set; }

public string Description { get; set; }

public float Price { get; set; }

public int StockCount { get; set; }

}

```

Now, we can register the entity cache services to [dependency injection](Dependency-Injection.md) in the `ConfigureServices` method of your [module class](Module-Development-Basics.md) with three generic parameters, as shown below:

```csharp

context.Services.AddEntityCache<Product, ProductDto, Guid>();

```

Since the entity cache system will perform the [object mapping](Object-To-Object-Mapping.md) (from `Product` to `ProductDto`), we should configure the object map. Here, an example configuration with [AutoMapper](https://automapper.org/):

```csharp

public class MyMapperProfile : Profile

{

public MyMapperProfile()

{

CreateMap<Product, ProductDto>();

}

}

```

Now, you can inject the `IEntityCache<ProductDto, Guid>` service wherever you want:

```csharp

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IEntityCache<ProductDto, Guid> \_productCache;

public ProductAppService(IEntityCache<ProductDto, Guid> productCache)

{

\_productCache = productCache;

}

public async Task<ProductDto> GetAsync(Guid id)

{

return await \_productCache.GetAsync(id);

}

}

```

Notice that the `\_productCache.GetAsync` method already returns a `ProductDto` object, so we could directly return it from out application service.

## Configuration

All of the `context.Services.AddEntityCache()` methods get an optional `DistributedCacheEntryOptions` parameter where you can easily configure the caching options:

```csharp

context.Services.AddEntityCache<Product, ProductDto, Guid>(

new DistributedCacheEntryOptions

{

SlidingExpiration = TimeSpan.FromMinutes(30)

}

);

```

> The default cache duration is **\*\*2 minutes\*\*** with the `AbsoluteExpirationRelativeToNow` configuration.

## Additional Notes

\* Entity classes should be serializable/deserializable to/from JSON to be cached (because it's serialized to JSON when saving in the [Distributed Cache](Caching.md)). If your entity class is not serializable, you can consider using a cache-item/DTO class instead, as explained before.

\* Entity Caching System is designed as **\*\*read-only\*\***. You should use the standard [repository](Repositories.md) methods to manipulate the entity if you need. If you need to manipulate (update) the entity, do not get it from the entity cache. Instead, read it from the repository, change it and update using the repository.

## See Also

\* [Distributed caching](Caching.md)

\* [Entities](Entities.md)

\* [Repositories](Repositories.md)

### Redis Cache

# Redis Cache

ABP Framework [Caching System](Caching.md) extends the [ASP.NET Core distributed cache](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed). So, **\*\*any provider\*\*** supported by the standard ASP.NET Core distributed cache can be usable in your application and can be configured just like **\*\*documented by Microsoft\*\***.

However, ABP provides an **\*\*integration package\*\*** for Redis Cache: [Volo.Abp.Caching.StackExchangeRedis](https://www.nuget.org/packages/Volo.Abp.Caching.StackExchangeRedis). There are two reasons for using this package, instead of the standard [Microsoft.Extensions.Caching.StackExchangeRedis](https://www.nuget.org/packages/Microsoft.Extensions.Caching.StackExchangeRedis/) package.

1. It implements `SetManyAsync` and `GetManyAsync` methods. These are not standard methods of the Microsoft Caching library, but added by the ABP Framework [Caching](Caching.md) system. They **\*\*significantly increases the performance\*\*** when you need to set/get multiple cache items with a single method call.

2. It **\*\*simplifies\*\*** the Redis cache **\*\*configuration\*\*** (will be explained below).

> Volo.Abp.Caching.StackExchangeRedis is already uses the Microsoft.Extensions.Caching.StackExchangeRedis package, but extends and improves it.

## Installation

> This package is already installed in the application startup template if it is using Redis.

Open a command line in the folder of your `.csproj` file and type the following ABP CLI command:

````bash

abp add-package Volo.Abp.Caching.StackExchangeRedis

````

## Configuration

Volo.Abp.Caching.StackExchangeRedis package automatically gets the Redis [configuration](Configuration.md) from the `IConfiguration`. So, for example, you can set your configuration inside the `appsettings.json`:

````js

"Redis": {

"IsEnabled": "true",

"Configuration": "127.0.0.1"

}

````

The setting `IsEnabled` is optional and will be considered `true` if it is not set.

Alternatively you can configure the standard [RedisCacheOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.caching.stackexchangeredis.rediscacheoptions) [options](Options.md) class in the `ConfigureServices` method of your [module](Module-Development-Basics.md):

````csharp

Configure<RedisCacheOptions>(options =>

{

//...

});

````

## See Also

\* [Caching](Caching.md)

## Configuration

# Configuration

ASP.NET Core has an flexible and extensible key-value based configuration system. In fact, the configuration system is a part of Microsoft.Extensions libraries and it is independent from ASP.NET Core. That means it can be used in any type of application. See [Microsoft's documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/) to learn the configuration infrastructure. ABP framework is 100% compatible with the configuration system.

## Connection Strings

# Connection Strings

> Connection string system is especially needed when you want to create or use a modular system. If you have a monolithic application with a single database, you can go with the [ABP startup solution template](Startup-Templates/Application.md), which is properly configured for you.

ABP Framework is designed to be [modular](Module-Development-Basics.md) and [multi-tenancy](Multi-Tenancy.md) aware. Connection string management is also designed to support these scenarios;

\* Allows to set separate connection strings for every module, so every module can have its own physical database. Modules even might be configured to use different database providers.

\* Allows to set separate connection string and use a separate database per tenant (in a SaaS application).

It also supports hybrid scenarios;

\* Allows to group modules into databases (e.g., all modules into a single shared database or two modules to database A, three modules to database B, one module to database C and rest of the modules to database D)

\* Allows to group tenants into databases, just like the modules.

\* Allows to separate databases per tenant per module (which might be hard to maintain for you because of too many databases, but the ABP framework supports it).

All the [pre-built application modules](Modules/Index.md) are designed to be compatible these scenarios.

## Configure the Connection Strings

See the following configuration:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=MyMainDb;Trusted\_Connection=True;",

"AbpIdentityServer": "Server=localhost;Database=MyIdsDb;Trusted\_Connection=True;",

"AbpPermissionManagement": "Server=localhost;Database=MyPermissionDb;Trusted\_Connection=True;"

}

````

> ABP uses the `IConfiguration` service to get the application configuration. While the simplest way to write configuration into the `appsettings.json` file, it is not limited to this file. You can use environment variables, user secrets, Azure Key Vault... etc. See the [configuration](Configuration.md) document for more.

This configuration defines three different connection strings:

\* `MyMainDb` (the `Default` connection string) is the main connection string of the application. If you don't specify a connection string for a module, it fallbacks to the `Default` connection string. The [application startup template](Startup-Templates/Application.md) is configured to use a single connection string, so all the modules uses a single, shared database.

\* `MyIdsDb` (the `AbpIdentityServer` connection string) is used by the [IdentityServer](Modules/IdentityServer.md) module.

\* `MyPermissionDb` (the `AbpPermissionManagement` connection string) is used by the [Permission Management](Modules/Permission-Management.md) module.

[Pre-built application modules](Modules/Index.md) define constants for the connection string names. For example, the [IdentityServer module](Modules/IdentityServer.md) defines a ` ConnectionStringName ` constant in the ` AbpIdentityServerDbProperties ` class (located in the ` Volo.Abp.IdentityServer ` namespace). Other modules similarly define constants, so you can investigate the connection string name.

### AbpDbConnectionOptions

`AbpDbConnectionOptions` is the options class that is used to set the connection strings and configure database structures.

#### Setting the connection strings

ABP uses the `AbpDbConnectionOptions` to get the connection strings. If you configure the connection strings as explained above, `AbpDbConnectionOptions` is automatically filled. However, you can set or override the connection strings using [the options pattern](Options.md). You can configure the `AbpDbConnectionOptions` in the `ConfigureServices` method of your [module](Module-Development-Basics.md) as shown below:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpDbConnectionOptions>(options =>

{

options.ConnectionStrings.Default = "...";

options.ConnectionStrings["AbpPermissionManagement"] = "...";

});

}

````

#### Configuring the database structures

`Databases` property of the `AbpDbConnectionOptions` class is used to group multiple connection strings (of multiple modules) to a single connection string.

See the following connection strings:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=MyMainDb;Trusted\_Connection=True;",

"AbpIdentity": "Server=localhost;Database=MySecondaryDb;Trusted\_Connection=True;",

"AbpIdentityServer": "Server=localhost;Database=MySecondaryDb;Trusted\_Connection=True;",

"AbpPermissionManagement": "Server=localhost;Database=MySecondaryDb;Trusted\_Connection=True;"

}

````

In this example, we've defined four connection strings, but the last three of them are the same; `AbpIdentity`, `AbpIdentityServer` and `AbpPermissionManagement` uses the same database, named `MySecondaryDb`. The main application and the other modules use the `Default` connection string, hence the `MyMainDb` database.

What we want to do here is to group three modules (`AbpIdentity`, `AbpIdentityServer` and `AbpPermissionManagement`) in a single database, but we needed to specify each one manually. Because the fallback connection string is the `Default` one, if we don't specify it for a module.

To eliminate the repetitive connection string definition, we can configure the `AbpDbConnectionOptions.Databases` property to group these connection strings, as shown in the following code (we place that in the `ConfigureServices` method of our [module class](Module-Development-Basics.md)):

````csharp

Configure<AbpDbConnectionOptions>(options =>

{

options.Databases.Configure("MySecondaryDb", db =>

{

db.MappedConnections.Add("AbpIdentity");

db.MappedConnections.Add("AbpIdentityServer");

db.MappedConnections.Add("AbpPermissionManagement");

});

});

````

Then we can change the `appsettings.json` file as shown in the following code block:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=MyMainDb;Trusted\_Connection=True;",

"MySecondaryDb": "Server=localhost;Database=MySecondaryDb;Trusted\_Connection=True;"

}

````

`MySecondaryDb` becomes the new connection string for the mapped connections.

> ABP first looks for the module-specific connection string, then looks if a database mapping is available, finally fallbacks to the `Default` connection string.

## Set the Connection String Name

A module typically has a unique connection string name associated to its `DbContext` class using the `ConnectionStringName` attribute. Example:

````csharp

[ConnectionStringName("AbpIdentityServer")]

public class IdentityServerDbContext

: AbpDbContext<IdentityServerDbContext>, IIdentityServerDbContext

{

}

````

For [Entity Framework Core](Entity-Framework-Core.md) and [MongoDB](MongoDB.md), write this to your `DbContext` class (and the interface if it has). In this way, ABP uses the specified connection string for the related `DbContext` instances.

## Database Migrations for the Entity Framework Core

Relational databases require to create the database and the database schema (tables, views... etc.) before using it.

The startup template (with EF Core ORM) comes with a single database and a `.EntityFrameworkCore` project that contains related classes and the migration files for that database. This project mainly defines a `YourProjectNameDbContext` class that calls the `Configure...()` methods of the used modules, like `builder.ConfigurePermissionManagement()`.

Once you want to separate a module's database, you typically will need to create a second migration path. See the [EF Core Migrations](Entity-Framework-Core-Migrations.md) document to learn how to create and use a different database for a desired module.

## Multi-Tenancy

See [the multi-tenancy document](Multi-Tenancy.md) to learn how to use separate databases for tenants.

## Replace the Connection String Resolver

ABP defines the `IConnectionStringResolver` and uses it whenever it needs a connection string. It has two pre-built implementations:

\* `DefaultConnectionStringResolver` uses the `AbpDbConnectionOptions` to select the connection string based on the rules defined in the "Configure the Connection Strings" section above.

\* `MultiTenantConnectionStringResolver` used for multi-tenant applications and tries to get the configured connection string for the current tenant if available. It uses the `ITenantStore` to find the connection strings. It inherits from the `DefaultConnectionStringResolver` and fallbacks to the base logic if no connection string specified for the current tenant.

If you need a custom logic to determine the connection string, implement the `IConnectionStringResolver` interface (optionally derive from the existing implementations) and replace the existing implementation using the [dependency injection](Dependency-Injection.md) system.

## Dependency Injection

# Dependency Injection

ABP's Dependency Injection system is developed based on Microsoft's [dependency injection extension](https://medium.com/volosoft/asp-net-core-dependency-injection-best-practices-tips-tricks-c6e9c67f9d96) library (Microsoft.Extensions.DependencyInjection nuget package). So, it's documentation is valid in ABP too.

> While ABP has no core dependency to any 3rd-party DI provider. However, it's required to use a provider that supports dynamic proxying and some other advanced features to make some ABP features properly work. Startup templates come with [Autofac](https://autofac.org/) installed. See [Autofac integration](Autofac-Integration.md) document for more information.

## Modularity

Since ABP is a modular framework, every module defines it's own services and registers via dependency injection in it's own separate [module class](Module-Development-Basics.md). Example:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//register dependencies here

}

}

````

## Conventional Registration

ABP introduces conventional service registration. You need not do anything to register a service by convention. It's automatically done. If you want to disable it, you can set `SkipAutoServiceRegistration` to `true` in the constructor of your module class. Example:

````C#

public class BlogModule : AbpModule

{

public BlogModule()

{

SkipAutoServiceRegistration = true;

}

}

````

Once you skip the auto registration, you should manually register your services. In that case, ``AddAssemblyOf`` extension method can help you to register all your services by convention. Example:

````c#

public class BlogModule : AbpModule

{

public BlogModule()

{

SkipAutoServiceRegistration = true;

}

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAssemblyOf<BlogModule>();

}

}

````

The sections below explain the conventions and configurations.

### Inherently Registered Types

Some specific types are registered to dependency injection by default. Examples:

\* Module classes are registered as singleton.

\* MVC controllers (inherit ``Controller`` or ``AbpController``) are registered as transient.

\* MVC page models (inherit ``PageModel`` or ``AbpPageModel``) are registered as transient.

\* MVC view components (inherit ``ViewComponent`` or ``AbpViewComponent``) are registered as transient.

\* Application services (inherit ``ApplicationService`` class or its subclasses) are registered as transient.

\* Repositories (implement ``BasicRepositoryBase`` class or its subclasses) are registered as transient.

\* Domain services (implement ``IDomainService`` interface or inherit ``DomainService`` class) are registered as transient.

Example:

````C#

public class BlogPostAppService : ApplicationService

{

}

````

``BlogPostAppService`` is automatically registered with transient lifetime since it's derived from a known base class.

### Dependency Interfaces

If you implement these interfaces, your class is registered to dependency injection automatically:

\* ``ITransientDependency`` to register with transient lifetime.

\* ``ISingletonDependency`` to register with singleton lifetime.

\* ``IScopedDependency`` to register with scoped lifetime.

Example:

````C#

public class TaxCalculator : ITransientDependency

{

}

````

``TaxCalculator`` is automatically registered with a transient lifetime since it implements ``ITransientDependency``.

### Dependency Attribute

Another way of configuring a service for dependency injection is to use ``DependencyAttribute``. It has the following properties:

\* ``Lifetime``: Lifetime of the registration: ``Singleton``, ``Transient`` or ``Scoped``.

\* ``TryRegister``: Set ``true`` to register the service only if it's not registered before. Uses TryAdd... extension methods of IServiceCollection.

\* ``ReplaceServices``: Set ``true`` to replace services if they are already registered before. Uses Replace extension method of IServiceCollection.

Example:

````C#

[Dependency(ServiceLifetime.Transient, ReplaceServices = true)]

public class TaxCalculator

{

}

````

``Dependency`` attribute has a higher priority than other dependency interfaces if it defines the ``Lifetime`` property.

### ExposeServices Attribute

``ExposeServicesAttribute`` is used to control which services are provided by the related class. Example:

````C#

[ExposeServices(typeof(ITaxCalculator))]

public class TaxCalculator: ICalculator, ITaxCalculator, ICanCalculate, ITransientDependency

{

}

````

``TaxCalculator`` class only exposes ``ITaxCalculator`` interface. That means you can only inject ``ITaxCalculator``, but can not inject ``TaxCalculator`` or ``ICalculator`` in your application.

### Exposed Services by Convention

If you do not specify which services to expose, ABP expose services by convention. So taking the ``TaxCalculator`` defined above:

\* The class itself is exposed by default. That means you can inject it by ``TaxCalculator`` class.

\* Default interfaces are exposed by default. Default interfaces are determined by naming convention. In this example, ``ICalculator`` and ``ITaxCalculator`` are default interfaces of ``TaxCalculator``, but ``ICanCalculate`` is not. A generic interface (e.g. `ICalculator<string>`) is also considered as a default interface if the naming convention is satisfied.

### Combining All Together

Combining attributes and interfaces is possible as long as it's meaningful.

````C#

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(ITaxCalculator))]

public class TaxCalculator : ITaxCalculator, ITransientDependency

{

}

````

### Manually Registering

In some cases, you may need to register a service to the `IServiceCollection` manually, especially if you need to use custom factory methods or singleton instances. In that case, you can directly add services just as [Microsoft documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) describes. Example:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Register an instance as singleton

context.Services.AddSingleton<TaxCalculator>(new TaxCalculator(taxRatio: 0.18));

//Register a factory method that resolves from IServiceProvider

context.Services.AddScoped<ITaxCalculator>(

sp => sp.GetRequiredService<TaxCalculator>()

);

}

}

````

### Replace a Service

If you need to replace an existing service (defined by the ABP framework or another module dependency), you have two options;

1. Use the `Dependency` attribute of the ABP framework as explained above.

2. Use the `IServiceCollection.Replace` method of the Microsoft Dependency Injection library. Example:

````csharp

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Replacing the IConnectionStringResolver service

context.Services.Replace(

ServiceDescriptor.Transient<

IConnectionStringResolver,

MyConnectionStringResolver

>());

}

}

````

## Injecting Dependencies

There are three common ways of using a service that has already been registered.

### Constructor Injection

This is the most common way of injecting a service into a class. For example:

````C#

public class TaxAppService : ApplicationService

{

private readonly ITaxCalculator \_taxCalculator;

public TaxAppService(ITaxCalculator taxCalculator)

{

\_taxCalculator = taxCalculator;

}

public async Task DoSomethingAsync()

{

//...use \_taxCalculator...

}

}

````

``TaxAppService`` gets ``ITaxCalculator`` in it's constructor. The dependency injection system automatically provides the requested service at runtime.

Constructor injection is preffered way of injecting dependencies to a class. In that way, the class can not be constructed unless all constructor-injected dependencies are provided. Thus, the class explicitly declares it's required services.

### Property Injection

Property injection is not supported by Microsoft Dependency Injection library. However, ABP can integrate with 3rd-party DI providers ([Autofac](https://autofac.org/), for example) to make property injection possible. Example:

````C#

public class MyService : ITransientDependency

{

public ILogger<MyService> Logger { get; set; }

public MyService()

{

Logger = NullLogger<MyService>.Instance;

}

public async Task DoSomethingAsync()

{

//...use Logger to write logs...

}

}

````

For a property-injection dependency, you declare a public property with public setter. This allows the DI framework to set it after creating your class.

Property injected dependencies are generally considered as **\*\*optional\*\*** dependencies. That means the service can properly work without them. ``Logger`` is such a dependency, ``MyService`` can continue to work without logging.

To make the dependency properly optional, we generally set a default/fallback value to the dependency. In this sample, NullLogger is used as fallback. Thus, ``MyService`` can work but does not write logs if DI framework or you don't set Logger property after creating ``MyService``.

One restriction of property injection is that you cannot use the dependency in your constructor, since it's set after the object construction.

Property injection is also useful when you want to design a base class that has some common services injected by default. If you're going to use constructor injection, all derived classes should also inject depended services into their own constructors which makes development harder. However, be very careful using property injection for non-optional services as it makes it harder to clearly see the requirements of a class.

#### DisablePropertyInjection Attribute

You can use `[DisablePropertyInjection]` attribute on classes or their properties to disable property injection for the whole class or some specific properties.

````C#

// Disabling for all properties of the MyService class

[DisablePropertyInjection]

public class MyService : ITransientDependency

{

public ILogger<MyService> Logger { get; set; }

public ITaxCalculator TaxCalculator { get; set; }

}

// Disabling only for the TaxCalculator property

public class MyService : ITransientDependency

{

public ILogger<MyService> Logger { get; set; }

[DisablePropertyInjection]

public ITaxCalculator TaxCalculator { get; set; }

}

````

### Resolve Service from IServiceProvider

You may want to resolve a service directly from ``IServiceProvider``. In that case, you can inject `IServiceProvider` into your class and use the ``GetService`` or the `GetRequiredService` method as shown below:

````C#

public class MyService : ITransientDependency

{

private readonly ITaxCalculator \_taxCalculator;

public MyService(IServiceProvider serviceProvider)

{

\_taxCalculator = serviceProvider.GetRequiredService<ITaxCalculator>();

}

}

````

### Dealing with multiple implementations

You can register multiple implementations of the same service interface. Assume that you have an `IExternalLogger` interface with two implementations:

````csharp

public interface IExternalLogger

{

Task LogAsync(string logText);

}

public class ElasticsearchExternalLogger : IExternalLogger

{

public async Task LogAsync(string logText)

{

//TODO...

}

}

public class AzureExternalLogger : IExternalLogger

{

public Task LogAsync(string logText)

{

throw new System.NotImplementedException();

}

}

````

In this example, we haven't registered any of the implementation classes to the dependency injection system yet. So, if we try to inject the `IExternalLogger` interface, we get an error indicating that no implementation found.

If we register both of the `ElasticsearchExternalLogger` and `AzureExternalLogger` services for the `IExternalLogger` interface, and then try to inject the `IExternalLogger` interface, then the last registered implementation will be used.

An example service injecting the `IExternalLogger` interface:

````csharp

public class MyService : ITransientDependency

{

private readonly IExternalLogger \_externalLogger;

public MyService(IExternalLogger externalLogger)

{

\_externalLogger = externalLogger;

}

public async Task DemoAsync()

{

await \_externalLogger.LogAsync("Example log message...");

}

}

````

Here, as said before, we get the last registered implementation. However, how to determine the last registered implementation?

If we implement one of the dependency interfaces (e.g. `ITransientDependency`), then the registration order will be uncertain (it may depend on the namespaces of the classes). The *\*last registered implementation\** can be different than you expect. So, it is not suggested to use the dependency interfaces to register multiple implementations.

You can register your services in the `ConfigureServices` method of your module:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddTransient<IExternalLogger, ElasticsearchExternalLogger>();

context.Services.AddTransient<IExternalLogger, AzureExternalLogger>();

}

````

In this case, you get an `AzureExternalLogger` instance when you inject the `IExternalLogger` interface, because the last registered implementation is the `AzureExternalLogger` class.

When you have multiple implementation of an interface, you may want to work with all these implementations. Assume that you want to write log to all the external loggers. We can change the `MyService` implementation as the following:

````csharp

public class MyService : ITransientDependency

{

private readonly IEnumerable<IExternalLogger> \_externalLoggers;

public MyService(IEnumerable<IExternalLogger> externalLoggers)

{

\_externalLoggers = externalLoggers;

}

public async Task DemoAsync()

{

foreach (var externalLogger in \_externalLoggers)

{

await externalLogger.LogAsync("Example log message...");

}

}

}

````

In this example, we are injecting `IEnumerable<IExternalLogger>` instead of `IExternalLogger`, so we have a collection of the `IExternalLogger` implementations. Then we are using a `foreach` loop to write the same log text to all the `IExternalLogger` implementations.

If you are using `IServiceProvider` to resolve dependencies, then use its `GetServices` method to obtain a collection of the service implementations:

````csharp

IEnumerable<IExternalLogger> services = \_serviceProvider.GetServices<IExternalLogger>();

````

### Releasing/Disposing Services

If you used a constructor or property injection, you don't need to be concerned about releasing the service's resources. However, if you have resolved a service from ``IServiceProvider``, in some cases, you might need to take care about releasing the service resources.

ASP.NET Core releases all services at the end of a current HTTP request, even if you directly resolved from ``IServiceProvider`` (assuming you injected `IServiceProvider`). But, there are several cases where you may want to release/dispose manually resolved services:

\* Your code is executed outside of ASP.NET Core request and the executer hasn't handled the service scope.

\* You only have a reference to the root service provider.

\* You may want to immediately release & dispose services (for example, you may creating too many services with big memory usages and don't want to overuse the memory).

In any case, you can create a service scope block to safely and immediately release services:

````C#

using (var scope = \_serviceProvider.CreateScope())

{

var service1 = scope.ServiceProvider.GetService<IMyService1>();

var service2 = scope.ServiceProvider.GetService<IMyService2>();

}

````

Both services are released when the created scope is disposed (at the end of the `using` block).

### Cached Service Providers

ABP provides two special services to optimize resolving services from `IServiceProvider`. `ICachedServiceProvider` and `ITransientCachedServiceProvider` both inherits from the `IServiceProvider` interface and internally caches the resolved services, so you get the same service instance even if you resolve a service multiple times.

The main difference is the `ICachedServiceProvider` is itself registered as scoped, while the `ITransientCachedServiceProvider` is registered as transient to the dependency injection system.

The following example injects the `ICachedServiceProvider` service and resolves a service in the `DoSomethingAsync` method:

````csharp

public class MyService : ITransientDependency

{

private readonly ICachedServiceProvider \_serviceProvider;

public MyService(ICachedServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public async Task DoSomethingAsync()

{

var taxCalculator = \_serviceProvider.GetRequiredService<ITaxCalculator>();

// TODO: Use the taxCalculator

}

}

````

With such a usage, you don't need to deal with creating service scopes and disposing the resolved services (as explained in the *\*Releasing/Disposing Services\** section above). Because all the services resolved from the `ICachedServiceProvider` will be released once the service scope of the `MyService` instance is disposed. Also, you don't need to care about memory leaks (because of creating too many `ITaxCalculator` instances if we call `DoSomethingAsync` too many times), because only one `ITaxCalculator` instance is created, and it is reused.

Since `ICachedServiceProvider` and `ITransientCachedServiceProvider` extends the standard `IServiceProvider` interface, you can use all the extension method of the `IServiceProvider` interface on them. In addition, they provides some other methods to provide a default value or a factory method for the services that are not found (that means not registered to the dependency injection system). Notice that the default value (or the value returned from your factory method) is also cached and reused.

Use `ICachedServiceProvider` (instead of `ITransientCachedServiceProvider`) unless you need to create the service cache per usage. `ITransientCachedServiceProvider` guarantees that the created service instances are not shared with any other service, even they are in the same service scope. The services resolved from `ICachedServiceProvider` are shared with other services in the same service scope (in the same HTTP Request, for example), so it can be thought as more optimized.

> ABP Framework also provides the `IAbpLazyServiceProvider` service. It does exists for backward compatibility and works exactly same with the `ITransientCachedServiceProvider` service. So, use the `ITransientCachedServiceProvider` since the `IAbpLazyServiceProvider` might be removed in future ABP versions.

## Advanced Features

### IServiceCollection.OnRegistered Event

You may want to perform an action for every service registered to the dependency injection. In the `PreConfigureServices` method of your module, register a callback using the `OnRegistered` method as shown below:

````csharp

public class AppModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

context.Services.OnRegistered(ctx =>

{

var type = ctx.ImplementationType;

//...

});

}

}

````

`ImplementationType` provides the service type. This callback is generally used to add interceptor to a service. Example:

````csharp

public class AppModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

context.Services.OnRegistered(ctx =>

{

if (ctx.ImplementationType.IsDefined(typeof(MyLogAttribute), true))

{

ctx.Interceptors.TryAdd<MyLogInterceptor>();

}

});

}

}

````

This example simply checks if the service class has `MyLogAttribute` attribute and adds `MyLogInterceptor` to the interceptor list if so.

> Notice that `OnRegistered` callback might be called multiple times for the same service class if it exposes more than one service/interface. So, it's safe to use `Interceptors.TryAdd` method instead of `Interceptors.Add` method. See [the documentation](Dynamic-Proxying-Interceptors.md) of dynamic proxying / interceptors.

## 3rd-Party Providers

While ABP has no core dependency to any 3rd-party DI provider, it's required to use a provider that supports dynamic proxying and some other advanced features to make some ABP features properly work.

Startup templates come with Autofac installed. See [Autofac integration](Autofac-Integration.md) document for more information.

## See Also

\* [ASP.NET Core Dependency Injection Best Practices, Tips & Tricks](https://medium.com/volosoft/asp-net-core-dependency-injection-best-practices-tips-tricks-c6e9c67f9d96)

### AutoFac Integration

# Autofac Integration

[Autofac](https://autofac.org/) is one of the most used dependency injection frameworks for .NET. It provides advanced features compared to .Net Core's standard DI library, like dynamic proxying and property injection.

## Install Autofac Integration

> All the [startup templates](Startup-Templates/Index.md) and samples are Autofac integrated. So, most of the time you don't need to manually install this package.

If you're not using a startup template, you can use the [ABP CLI](CLI.md) to install it to your project. Execute the following command in the folder that contains the .csproj file of your project (suggested to add it to the executable/web project):

````bash

abp add-package Volo.Abp.Autofac

````

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Autofac).

>

Finally, configure `AbpApplicationCreationOptions` to replace default dependency injection services by Autofac. It depends on the application type.

### ASP.NET Core Application

Call `UseAutofac()` in the **\*\*Program.cs\*\*** file as shown below:

````csharp

public class Program

{

public static int Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

internal static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

})

.UseAutofac(); //Integrate Autofac!

}

````

If you are using the static `WebApplication` class, you can call the `UseAutofac()` extension method as shown below:

````csharp

public class Program

{

public async static Task Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

builder.Host.UseAutofac(); // Integrate Autofac!

await builder.AddApplicationAsync<MyProjectNameWebModule>();

var app = builder.Build();

await app.InitializeApplicationAsync();

await app.RunAsync();

}

}

````

### Console Application

Call `UseAutofac()` method in the `AbpApplicationFactory.Create` options as shown below:

````csharp

using System;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp;

namespace AbpConsoleDemo

{

class Program

{

static void Main(string[] args)

{

using (var application = AbpApplicationFactory.Create<AppModule>(options =>

{

options.UseAutofac(); //Autofac integration

}))

{

//...

}

}

}

}

````

## Using the Autofac Registration API

If you want to use Autofac's advanced [registration API](https://autofac.readthedocs.io/en/latest/register/registration.html), you need to access the `ContainerBuilder` object. [Volo.Abp.Autofac](https://www.nuget.org/packages/Volo.Abp.Autofac) nuget package defines the `IServiceCollection.GetContainerBuilder()` extension method to obtain the `ContainerBuilder` object.

**\*\*Example: Get the** `ContainerBuilder` **object in the** `ConfigureServices` **method of your [**module class**](Module-Development-Basics.md)\*\***

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var containerBuilder = context.Services.GetContainerBuilder();

containerBuilder.RegisterType<MyService>(); // Using Autofac's registration API

}

````

> You should install the [Volo.Abp.Autofac](https://www.nuget.org/packages/Volo.Abp.Autofac) nuget package to the project that you want to use the Autofac API.

## Exception Handling

# Exception Handling

ABP provides a built-in infrastructure and offers a standard model for handling exceptions.

\* Automatically **\*\*handles all exceptions\*\*** and sends a standard **\*\*formatted error message\*\*** to the client for an API/AJAX request.

\* Automatically hides **\*\*internal infrastructure errors\*\*** and returns a standard error message.

\* Provides an easy and configurable way to **\*\*localize\*\*** exception messages.

\* Automatically maps standard exceptions to **\*\*HTTP status codes\*\*** and provides a configurable option to map custom exceptions.

## Automatic Exception Handling

`AbpExceptionFilter` handles an exception if **\*\*any of the following conditions\*\*** are met:

\* Exception is thrown by a **\*\*controller action\*\*** which returns an **\*\*object result\*\*** (not a view result).

\* The request is an AJAX request (`X-Requested-With` HTTP header value is `XMLHttpRequest`).

\* Client explicitly accepts the `application/json` content type (via `accept` HTTP header).

If the exception is handled it's automatically **\*\*logged\*\*** and a formatted **\*\*JSON message\*\*** is returned to the client.

### Error Message Format

Error Message is an instance of the `RemoteServiceErrorResponse` class. The simplest error JSON has a **\*\*message\*\*** property as shown below:

````json

{

"error": {

"message": "This topic is locked and can not add a new message"

}

}

````

There are **\*\*optional fields\*\*** those can be filled based upon the exception that has occurred.

##### Error Code

Error **\*\*code\*\*** is an optional and unique string value for the exception. Thrown `Exception` should implement the `IHasErrorCode` interface to fill this field. Example JSON value:

````json

{

"error": {

"code": "App:010042",

"message": "This topic is locked and can not add a new message"

}

}

````

Error code can also be used to localize the exception and customize the HTTP status code (see the related sections below).

##### Error Details

Error **\*\*details\*\*** in an optional field of the JSON error message. Thrown `Exception` should implement the `IHasErrorDetails` interface to fill this field. Example JSON value:

```json

{

"error": {

"code": "App:010042",

"message": "This topic is locked and can not add a new message",

"details": "A more detailed info about the error..."

}

}

```

##### Validation Errors

**\*\*validationErrors\*\*** is a standard field that is filled if the thrown exception implements the `IHasValidationErrors` interface.

````json

{

"error": {

"code": "App:010046",

"message": "Your request is not valid, please correct and try again!",

"validationErrors": [{

"message": "Username should be minimum length of 3.",

"members": ["userName"]

},

{

"message": "Password is required",

"members": ["password"]

}]

}

}

````

`AbpValidationException` implements the `IHasValidationErrors` interface and it is automatically thrown by the framework when a request input is not valid. So, usually you don't need to deal with validation errors unless you have higly customised validation logic.

### Logging

Caught exceptions are automatically logged.

#### Log Level

Exceptions are logged with the `Error` level by default. The Log level can be determined by the exception if it implements the `IHasLogLevel` interface. Example:

````C#

public class MyException : Exception, IHasLogLevel

{

public LogLevel LogLevel { get; set; } = LogLevel.Warning;

//...

}

````

#### Self Logging Exceptions

Some exception types may need to write additional logs. They can implement the `IExceptionWithSelfLogging` if needed. Example:

````C#

public class MyException : Exception, IExceptionWithSelfLogging

{

public void Log(ILogger logger)

{

//...log additional info

}

}

````

> `ILogger.LogException` extension methods is used to write exception logs. You can use the same extension method when needed.

## Business Exceptions

Most of your own exceptions will be business exceptions. The `IBusinessException` interface is used to mark an exception as a business exception.

`BusinessException` implements the `IBusinessException` interface in addition to the `IHasErrorCode`, `IHasErrorDetails` and `IHasLogLevel` interfaces. The default log level is `Warning`.

Usually you have an error code related to a particular business exception. For example:

````C#

throw new BusinessException(QaErrorCodes.CanNotVoteYourOwnAnswer);

````

`QaErrorCodes.CanNotVoteYourOwnAnswer` is just a `const string`. The following error code format is recommended:

````

<code-namespace>:<error-code>

````

**\*\*code-namespace\*\*** is a **\*\*unique value\*\*** specific to your module/application. Example:

````

Volo.Qa:010002

````

`Volo.Qa` is the code-namespace here. code-namespace is then will be used while **\*\*localizing\*\*** exception messages.

\* You can **\*\*directly throw\*\*** a `BusinessException` or **\*\*derive\*\*** your own exception types from it when needed.

\* All properties are optional for the `BusinessException` class. But you generally set either `ErrorCode` or `Message` property.

## Exception Localization

One problem with throwing exceptions is how to localize error messages while sending it to the client. ABP offers two models and their variants.

### User Friendly Exception

If an exception implements the `IUserFriendlyException` interface, then ABP does not change it's `Message` and `Details` properties and directly send it to the client.

`UserFriendlyException` class is the built-in implementation of the `IUserFriendlyException` interface. Example usage:

````C#

throw new UserFriendlyException(

"Username should be unique!"

);

````

In this way, there is **\*\*no need for localization\*\*** at all. If you want to localize the message, you can inject and use the standard **\*\*string localizer\*\*** (see the [localization document](Localization.md)). Example:

````C#

throw new UserFriendlyException(\_stringLocalizer["UserNameShouldBeUniqueMessage"]);

````

Then define it in the **\*\*localization resource\*\*** for each language. Example:

````json

{

"culture": "en",

"texts": {

"UserNameShouldBeUniqueMessage": "Username should be unique!"

}

}

````

String localizer already supports **\*\*parameterized messages\*\***. For example:

````C#

throw new UserFriendlyException(\_stringLocalizer["UserNameShouldBeUniqueMessage", "john"]);

````

Then the localization text can be:

````json

"UserNameShouldBeUniqueMessage": "Username should be unique! '{0}' is already taken!"

````

\* The `IUserFriendlyException` interface is derived from the `IBusinessException` and the `UserFriendlyException` class is derived from the `BusinessException` class.

### Using Error Codes

`UserFriendlyException` is fine, but it has a few problems in advanced usages:

\* It requires you to **\*\*inject the string localizer\*\*** everywhere and always use it while throwing exceptions.

\* However, in some of the cases, it may **\*\*not be possible\*\*** to inject the string localizer (in a static context or in an entity method).

Instead of localizing the message while throwing the exception, you can separate the process using **\*\*error codes\*\***.

First, define the **\*\*code-namespace\*\*** to **\*\*localization resource\*\*** mapping in the module configuration:

````C#

services.Configure<AbpExceptionLocalizationOptions>(options =>

{

options.MapCodeNamespace("Volo.Qa", typeof(QaResource));

});

````

Then any of the exceptions with `Volo.Qa` namespace will be localized using their given localization resource. The localization resource should always have an entry with the error code key. Example:

````json

{

"culture": "en",

"texts": {

"Volo.Qa:010002": "You can not vote your own answer!"

}

}

````

Then a business exception can be thrown with the error code:

````C#

throw new BusinessException(QaDomainErrorCodes.CanNotVoteYourOwnAnswer);

````

\* Throwing any exception implementing the `IHasErrorCode` interface behaves the same. So, the error code localization approach is not unique to the `BusinessException` class.

\* Defining localized string is not required for an error message. If it's not defined, ABP sends the default error message to the client. It does not use the `Message` property of the exception! if you want that, use the `UserFriendlyException` (or use an exception type that implements the `IUserFriendlyException` interface).

#### Using Message Parameters

If you have a parameterized error message, then you can set it with the exception's `Data` property. For example:

````C#

throw new BusinessException("App:010046")

{

Data =

{

{"UserName", "john"}

}

};

````

Fortunately there is a shortcut way to code this:

````C#

throw new BusinessException("App:010046")

.WithData("UserName", "john");

````

Then the localized text can contain the `UserName` parameter:

````json

{

"culture": "en",

"texts": {

"App:010046": "Username should be unique. '{UserName}' is already taken!"

}

}

````

\* `WithData` can be chained with more than one parameter (like `.WithData(...).WithData(...)`).

## HTTP Status Code Mapping

ABP tries to automatically determine the most suitable HTTP status code for common exception types by following these rules:

\* For the `AbpAuthorizationException`:

\* Returns `401` (unauthorized) if user has not logged in.

\* Returns `403` (forbidden) if user has logged in.

\* Returns `400` (bad request) for the `AbpValidationException`.

\* Returns `404` (not found) for the `EntityNotFoundException`.

\* Returns `403` (forbidden) for the `IBusinessException` (and `IUserFriendlyException` since it extends the `IBusinessException`).

\* Returns `501` (not implemented) for the `NotImplementedException`.

\* Returns `500` (internal server error) for other exceptions (those are assumed as infrastructure exceptions).

The `IHttpExceptionStatusCodeFinder` is used to automatically determine the HTTP status code. The default implementation is the `DefaultHttpExceptionStatusCodeFinder` class. It can be replaced or extended as needed.

### Custom Mappings

Automatic HTTP status code determination can be overrided by custom mappings. For example:

````C#

services.Configure<AbpExceptionHttpStatusCodeOptions>(options =>

{

options.Map("Volo.Qa:010002", HttpStatusCode.Conflict);

});

````

## Subscribing to the Exceptions

It is possible to be informed when the ABP Framework **\*\*handles an exception\*\***. It automatically **\*\*logs\*\*** all the exceptions to the standard [logger](Logging.md), but you may want to do more.

In this case, create a class derived from the `ExceptionSubscriber` class in your application:

````csharp

public class MyExceptionSubscriber : ExceptionSubscriber

{

public async override Task HandleAsync(ExceptionNotificationContext context)

{

//TODO...

}

}

````

The `context` object contains necessary information about the exception occurred.

> You can have multiple subscribers, each gets a copy of the exception. Exceptions thrown by your subscriber is ignored (but still logged).

## Built-In Exceptions

Some exception types are automatically thrown by the framework:

- `AbpAuthorizationException` is thrown if the current user has no permission to perform the requested operation. See [authorization](Authorization.md) for more.

- `AbpValidationException` is thrown if the input of the current request is not valid. See [validation](Validation.md) for more.

- `EntityNotFoundException` is thrown if the requested entity is not available. This is mostly thrown by [repositories](Repositories.md).

You can also throw these type of exceptions in your code (although it's rarely needed).

## AbpExceptionHandlingOptions

`AbpExceptionHandlingOptions` is the main [options object](Options.md) to configure the exception handling system. You can configure it in the `ConfigureServices` method of your [module](Module-Development-Basics.md):

````csharp

Configure<AbpExceptionHandlingOptions>(options =>

{

options.SendExceptionsDetailsToClients = true;

options.SendStackTraceToClients = false;

});

````

Here, a list of the options you can configure:

\* `SendExceptionsDetailsToClients` (default: `false`): You can enable or disable sending exception details to the client.

\* `SendStackTraceToClients` (default: `true`): You can enable or disable sending the stack trace of exception to the client. If you want to send the stack trace to the client, you must set both `SendStackTraceToClients` and `SendExceptionsDetailsToClients` options to `true` otherwise, the stack trace will not be sent to the client.

## Localization

# Localization

ABP's localization system is seamlessly integrated to the `Microsoft.Extensions.Localization` package and compatible with the [Microsoft's localization documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization). It adds some useful features and enhancements to make it easier to use in real life application scenarios.

## Installation

> This package is already installed by default with the startup template. So, most of the time, you don't need to install it manually.

You can use the [ABP CLI](CLI.md) to install the Volo.Abp.Localization package to your project. Execute the following command in the folder of the .csproj file that you want to install the package on:

```bash

abp add-package Volo.Abp.Localization

```

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Localization).

Then you can add **\*\*AbpLocalizationModule\*\*** dependency to your module:

```c#

using Volo.Abp.Modularity;

using Volo.Abp.Localization;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpLocalizationModule))]

public class MyModule : AbpModule

{

//...

}

}

```

## Creating A Localization Resource

A localization resource is used to group related localization strings together and separate them from other localization strings of the application. A [module](Module-Development-Basics.md) generally defines its own localization resource. Localization resource is just a plain class. Example:

````C#

public class TestResource

{

}

````

Then it should be added using `AbpLocalizationOptions` as shown below:

````C#

[DependsOn(typeof(AbpLocalizationModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpVirtualFileSystemOptions>(options =>

{

// "YourRootNameSpace" is the root namespace of your project. It can be empty if your root namespace is empty.

options.FileSets.AddEmbedded<MyModule>("YourRootNameSpace");

});

Configure<AbpLocalizationOptions>(options =>

{

//Define a new localization resource (TestResource)

options.Resources

.Add<TestResource>("en")

.AddVirtualJson("/Localization/Resources/Test");

});

}

}

````

In this example;

\* Added a new localization resource with "en" (English) as the default culture.

\* Used JSON files to store the localization strings.

\* JSON files are embedded into the assembly using `AbpVirtualFileSystemOptions` (see [virtual file system](Virtual-File-System.md)).

JSON files are located under "/Localization/Resources/Test" project folder as shown below:

![localization-resource-json-files](images/localization-resource-json-files.png)

A JSON localization file content is shown below:

````json

{

"culture": "en",

"texts": {

"HelloWorld": "Hello World!"

}

}

````

\* Every localization file should define the `culture` code for the file (like "en" or "en-US").

\* `texts` section just contains key-value collection of the localization strings (keys may have spaces too).

> ABP will ignore (skip) the JSON file if the `culture` section is missing.

### Default Resource

`AbpLocalizationOptions.DefaultResourceType` can be set to a resource type, so it is used when the localization resource was not specified:

````csharp

Configure<AbpLocalizationOptions>(options =>

{

options.DefaultResourceType = typeof(TestResource);

});

````

> The [application startup template](Startup-Templates/Application.md) sets `DefaultResourceType` to the localization resource of the application.

### Short Localization Resource Name

Localization resources are also available in the client (JavaScript) side. So, setting a short name for the localization resource makes it easy to use localization texts. Example:

````C#

[LocalizationResourceName("Test")]

public class TestResource

{

}

````

See the Getting Localized Test / Client Side section below.

### Inherit From Other Resources

A resource can inherit from other resources which makes possible to re-use existing localization strings without referring the existing resource. Example:

````C#

[InheritResource(typeof(AbpValidationResource))]

public class TestResource

{

}

````

Alternative inheritance by configuring the `AbpLocalizationOptions`:

````C#

services.Configure<AbpLocalizationOptions>(options =>

{

options.Resources

.Add<TestResource>("en") //Define the resource by "en" default culture

.AddVirtualJson("/Localization/Resources/Test") //Add strings from virtual json files

.AddBaseTypes(typeof(AbpValidationResource)); //Inherit from an existing resource

});

````

\* A resource may inherit from multiple resources.

\* If the new resource defines the same localized string, it overrides the string.

### Extending Existing Resource

Inheriting from a resource creates a new resource without modifying the existing one. In some cases, you may want to not create a new resource but directly extend an existing resource. Example:

````C#

services.Configure<AbpLocalizationOptions>(options =>

{

options.Resources

.Get<TestResource>()

.AddVirtualJson("/Localization/Resources/Test/Extensions");

});

````

\* If an extension file defines the same localized string, it overrides the string.

## Getting the Localized Texts

Getting the localized text is pretty standard.

### Simplest Usage In A Class

Just inject the `IStringLocalizer<TResource>` service and use it like shown below:

````csharp

public class MyService : ITransientDependency

{

private readonly IStringLocalizer<TestResource> \_localizer;

public MyService(IStringLocalizer<TestResource> localizer)

{

\_localizer = localizer;

}

public void Foo()

{

var str = \_localizer["HelloWorld"];

}

}

````

##### Format Arguments

Format arguments can be passed after the localization key. If your message is `Hello {0}, welcome!`, then you can pass the `{0}` argument to the localizer like `\_localizer["HelloMessage", "John"]`.

> Refer to the [Microsoft's localization documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization) for details about using the localization.

### Using In A Razor View/Page

Use `IHtmlLocalizer<T>` in razor views/pages;

````c#

@inject IHtmlLocalizer<TestResource> Localizer

<h1>@Localizer["HelloWorld"]</h1>

````

### Special Base Classes

Some ABP Framework base classes provide a `L` property to use the localizer even easier.

**\*\*Example: Localize a text in an application service method\*\***

```csharp

using System.Threading.Tasks;

using MyProject.Localization;

using Volo.Abp.Application.Services;

namespace MyProject

{

public class TestAppService : ApplicationService

{

public TestAppService()

{

LocalizationResource = typeof(MyProjectResource);

}

public async Task DoIt()

{

var str = L["HelloWorld"];

}

}

}

```

When you set the `LocalizationResource` in the constructor, the `ApplicationService` class uses that resource type when you use the `L` property, just like in the `DoIt()` method.

Setting `LocalizationResource` in every application service can be tedious. You can create an abstract base application service class, set it there and derive your application services from that base class. This is already implemented when you create a new project with the [startup templates](Startup-Templates/Application.md). So, you can simply inherit from the base class directly use the `L` property:

```csharp

using System.Threading.Tasks;

namespace MyProject

{

public class TestAppService : MyProjectAppService

{

public async Task DoIt()

{

var str = L["HelloWorld"];

}

}

}

```

The `L` property is also available for some other base classes like `AbpController` and `AbpPageModel`.

## The Client Side

See the following documents to learn how to reuse the same localization texts in the JavaScript side;

\* [Localization for the MVC / Razor Pages UI](UI/AspNetCore/JavaScript-API/Localization.md)

\* [Localization for the Blazor UI](UI/Blazor/Localization.md)

\* [Localization for the Angular UI](UI/Angular/Localization.md)

## Logging

# Logging

ABP Framework doesn't implement any logging infrastructure. It uses the [ASP.NET Core's logging system](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging).

> .NET Core's logging system is actually independent from the ASP.NET Core. It is usable in any type of application.

## Object Extensions

# Object Extensions

ABP Framework provides an **\*\*object extension system\*\*** to allow you to **\*\*add extra properties\*\*** to an existing object **\*\*without modifying\*\*** the related class. This allows to extend functionalities implemented by a depended [application module](Modules/Index.md), especially when you want to [extend entities](Customizing-Application-Modules-Extending-Entities.md) and [DTOs](Customizing-Application-Modules-Overriding-Services.md) defined by the module.

> Object extension system normally is not needed for your own objects since you can easily add regular properties to your own classes.

## IHasExtraProperties Interface

This is the interface to make a class extensible. It simply defines a `Dictionary` property:

````csharp

ExtraPropertyDictionary ExtraProperties { get; }

````

`ExtraPropertyDictionary` class is inherited from the `Dictionary<string, object>` class. You can add or get extra properties using this dictionary.

### Base Classes

`IHasExtraProperties` interface is implemented by several base classes by default:

\* Implemented by the `AggregateRoot` class (see [entities](Entities.md)).

\* Implemented by `ExtensibleEntityDto`, `ExtensibleAuditedEntityDto`... base [DTO](Data-Transfer-Objects.md) classes.

\* Implemented by the `ExtensibleObject`, which is a simple base class can be inherited for any type of object.

So, if you inherit from these classes, your class will also be extensible. If not, you can always implement it manually.

### Fundamental Extension Methods

While you can directly use the `ExtraProperties` property of a class, it is suggested to use the following extension methods while working with the extra properties.

#### SetProperty

Used to set the value of an extra property:

````csharp

user.SetProperty("Title", "My Title");

user.SetProperty("IsSuperUser", true);

````

`SetProperty` returns the same object, so you can chain it:

````csharp

user.SetProperty("Title", "My Title")

.SetProperty("IsSuperUser", true);

````

#### GetProperty

Used to read the value of an extra property:

````csharp

var title = user.GetProperty<string>("Title");

if (user.GetProperty<bool>("IsSuperUser"))

{

//...

}

````

\* `GetProperty` is a generic method and takes the object type as the generic parameter.

\* Returns the default value if given property was not set before (default value is `0` for `int`, `false` for `bool`... etc).

##### Non Primitive Property Types

If your property type is not a primitive (int, bool, enum, string... etc) type, then you need to use non-generic version of the `GetProperty` which returns an `object`.

#### HasProperty

Used to check if the object has a property set before.

#### RemoveProperty

Used to remove a property from the object. Use this methods instead of setting a `null` value for the property.

### Some Best Practices

Using magic strings for the property names is dangerous since you can easily type the property name wrong - it is not type safe. Instead;

\* Define a constant for your extra property names

\* Create extension methods to easily set your extra properties.

Example:

````csharp

public static class IdentityUserExtensions

{

private const string TitlePropertyName = "Title";

public static void SetTitle(this IdentityUser user, string title)

{

user.SetProperty(TitlePropertyName, title);

}

public static string GetTitle(this IdentityUser user)

{

return user.GetProperty<string>(TitlePropertyName);

}

}

````

Then you can easily set or get the `Title` property:

````csharp

user.SetTitle("My Title");

var title = user.GetTitle();

````

## Object Extension Manager

While you can set arbitrary properties to an extensible object (which implements the `IHasExtraProperties` interface), `ObjectExtensionManager` is used to explicitly define extra properties for extensible classes.

Explicitly defining an extra property has some use cases:

\* Allows to control how the extra property is handled on object to object mapping (see the section below).

\* Allows to define metadata for the property. For example, you can map an extra property to a table field in the database while using the [EF Core](Entity-Framework-Core.md).

> `ObjectExtensionManager` implements the singleton pattern (`ObjectExtensionManager.Instance`) and you should define object extensions before your application startup. The [application startup template](Startup-Templates/Application.md) has some pre-defined static classes to safely define object extensions inside.

### AddOrUpdate

`AddOrUpdate` is the main method to define a extra properties or update extra properties for an object.

Example: Define extra properties for the `IdentityUser` entity:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdate<IdentityUser>(options =>

{

options.AddOrUpdateProperty<string>("SocialSecurityNumber");

options.AddOrUpdateProperty<bool>("IsSuperUser");

}

);

````

### AddOrUpdateProperty

While `AddOrUpdateProperty` can be used on the `options` as shown before, if you want to define a single extra property, you can use the shortcut extension method too:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>("SocialSecurityNumber");

````

Sometimes it would be practical to define a single extra property to multiple types. Instead of defining one by one, you can use the following code:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<string>(

new[]

{

typeof(IdentityUserDto),

typeof(IdentityUserCreateDto),

typeof(IdentityUserUpdateDto)

},

"SocialSecurityNumber"

);

````

### Property Configuration

`AddOrUpdateProperty` can also get an action that can perform additional configuration on the property definition:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

//Configure options...

});

````

> `options` has a dictionary, named `Configuration` which makes the object extension definitions even extensible. It is used by the EF Core to map extra properties to table fields in the database. See the [extending entities](Customizing-Application-Modules-Extending-Entities.md) document.

The following sections explain the fundamental property configuration options.

#### Default Value

A default value is automatically set for the new property, which is the natural default value for the property type, like `null` for `string`, `false` for `bool` or `0` for `int`.

There are two ways to override the default value:

##### DefaultValue Option

`DefaultValue` option can be set to any value:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, int>(

"MyIntProperty",

options =>

{

options.DefaultValue = 42;

});

````

##### DefaultValueFactory Options

`DefaultValueFactory` can be set to a function that returns the default value:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, DateTime>(

"MyDateTimeProperty",

options =>

{

options.DefaultValueFactory = () => DateTime.Now;

});

````

`options.DefaultValueFactory` has a higher priority than the `options.DefaultValue` .

> Tip: Use `DefaultValueFactory` option only if the default value may change over the time (like `DateTime.Now` in this example). If it is a constant value, then use the `DefaultValue` option.

#### CheckPairDefinitionOnMapping

Controls how to check property definitions while mapping two extensible objects. See the "Object to Object Mapping" section to understand the `CheckPairDefinitionOnMapping` option better.

## Validation

You may want to add some **\*\*validation rules\*\*** for the extra properties you've defined. `AddOrUpdateProperty` method options allows two ways of performing validation:

1. You can add **\*\*data annotation attributes\*\*** for a property.

2. You can write an action (code block) to perform a **\*\*custom validation\*\***.

Validation works when you use the object in a method that is **\*\*automatically validated\*\*** (e.g. controller actions, page handler methods, application service methods...). So, all extra properties are validated whenever the extended object is being validated.

### Data Annotation Attributes

All of the standard data annotation attributes are valid for extra properties. Example:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUserCreateDto, string>(

"SocialSecurityNumber",

options =>

{

options.Attributes.Add(new RequiredAttribute());

options.Attributes.Add(

new StringLengthAttribute(32) {

MinimumLength = 6

}

);

});

````

With this configuration, `IdentityUserCreateDto` objects will be invalid without a valid `SocialSecurityNumber` value provided.

#### Default Validation Attributes

There are some attributes **\*\*automatically added\*\*** when you create certain type of properties;

\* `RequiredAttribute` is added for non nullable primitive property types (e.g. `int`, `bool`, `DateTime`...) and `enum` types.

\* `EnumDataTypeAttribute` is added for enum types, to prevent to set invalid enum values.

Use `options.Attributes.Clear();` if you don't want these attributes.

### Custom Validation

If you need, you can add a custom action that is executed to validate the extra properties. Example:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUserCreateDto, string>(

"SocialSecurityNumber",

options =>

{

options.Validators.Add(context =>

{

var socialSecurityNumber = context.Value as string;

if (socialSecurityNumber == null ||

socialSecurityNumber.StartsWith("X"))

{

context.ValidationErrors.Add(

new ValidationResult(

"Invalid social security number: " + socialSecurityNumber,

new[] { "SocialSecurityNumber" }

)

);

}

});

});

````

`context.ServiceProvider` can be used to resolve a service dependency for advanced scenarios.

In addition to add custom validation logic for a single property, you can add a custom validation logic that is executed in object level. Example:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdate<IdentityUserCreateDto>(objConfig =>

{

//Define two properties with their own validation rules

objConfig.AddOrUpdateProperty<string>("Password", propertyConfig =>

{

propertyConfig.Attributes.Add(new RequiredAttribute());

});

objConfig.AddOrUpdateProperty<string>("PasswordRepeat", propertyConfig =>

{

propertyConfig.Attributes.Add(new RequiredAttribute());

});

//Write a common validation logic works on multiple properties

objConfig.Validators.Add(context =>

{

if (context.ValidatingObject.GetProperty<string>("Password") !=

context.ValidatingObject.GetProperty<string>("PasswordRepeat"))

{

context.ValidationErrors.Add(

new ValidationResult(

"Please repeat the same password!",

new[] { "Password", "PasswordRepeat" }

)

);

}

});

});

````

## Object to Object Mapping

Assume that you've added an extra property to an extensible entity object and used auto [object to object mapping](Object-To-Object-Mapping.md) to map this entity to an extensible DTO class. You need to be careful in such a case, because the extra property may contain a **\*\*sensitive data\*\*** that should not be available to clients.

This section offers some **\*\*good practices\*\*** to control your extra properties on object mapping.

### MapExtraPropertiesTo

`MapExtraPropertiesTo` is an extension method provided by the ABP Framework to copy extra properties from an object to another in a controlled manner. Example usage:

````csharp

identityUser.MapExtraPropertiesTo(identityUserDto);

````

`MapExtraPropertiesTo` **\*\*requires to define properties\*\*** (as described above) in **\*\*both sides\*\*** (`IdentityUser` and `IdentityUserDto` in this case) in order to copy the value to the target object. Otherwise, it doesn't copy the value even if it does exists in the source object (`identityUser` in this example). There are some ways to overload this restriction.

#### MappingPropertyDefinitionChecks

`MapExtraPropertiesTo` gets an additional parameter to control the definition check for a single mapping operation:

````csharp

identityUser.MapExtraPropertiesTo(

identityUserDto,

MappingPropertyDefinitionChecks.None

);

````

> Be careful since `MappingPropertyDefinitionChecks.None` copies all extra properties without any check. `MappingPropertyDefinitionChecks` enum has other members too.

If you want to completely disable definition check for a property, you can do it while defining the extra property (or update an existing definition) as shown below:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.CheckPairDefinitionOnMapping = false;

});

````

#### Ignored Properties

You may want to ignore some properties on a specific mapping operation:

````csharp

identityUser.MapExtraPropertiesTo(

identityUserDto,

ignoredProperties: new[] {"MySensitiveProp"}

);

````

Ignored properties are not copied to the target object.

#### AutoMapper Integration

If you're using the [AutoMapper](https://automapper.org/) library, the ABP Framework also provides an extension method to utilize the `MapExtraPropertiesTo` method defined above.

You can use the `MapExtraProperties()` method inside your mapping profile.

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<IdentityUser, IdentityUserDto>()

.MapExtraProperties();

}

}

````

It has the same parameters with the `MapExtraPropertiesTo` method.

## Entity Framework Core Database Mapping

If you're using the EF Core, you can map an extra property to a table field in the database. Example:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.MapEfCore(b => b.HasMaxLength(32));

}

);

````

See the [Entity Framework Core Integration document](Entity-Framework-Core.md) for more.

## See Also

\* [Module Entity Extensions](Module-Entity-Extensions.md)

## Options

# Options

Microsoft has introduced [the options pattern](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options) that is used to configure a group of settings used by the framework services. This pattern is implemented by the [Microsoft.Extensions.Options](https://www.nuget.org/packages/Microsoft.Extensions.Options) NuGet package, so it is usable by any type of applications in addition to ASP.NET Core based applications.

ABP framework follows this option pattern and defines options classes to configure the framework and the modules (they are explained in the documents of the related feature).

Since [the Microsoft documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options) explains the pattern in detail, no reason to repeat all. However, ABP adds a few more features and they will be explained here.

## Configure Options

You typically configure options in the `ConfigureServices` of the `Startup` class. However, since ABP framework provides a modular infrastructure, you configure options in the `ConfigureServices` of your [module](Module-Development-Basics.md). Example:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.Configure<AbpAuditingOptions>(options =>

{

options.IsEnabled = false;

});

}

````

\* `AbpAuditingOptions` is a simple class defines some properties like `IsEnabled` used here.

\* `AbpModule` base class defines `Configure` method to make the code simpler. So, instead of `context.Services.Configure<...>`, you can directly use the `Configure<...>` shortcut method.

If you are developing a reusable module, you may need to define an options class to allow developers to configure your module. In this case, define a plain options class as shown below:

````csharp

public class MyOptions

{

public int Value1 { get; set; }

public bool Value2 { get; set; }

}

````

Then developers can configure your options just like the `AbpAuditingOptions` example above:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<MyOptions>(options =>

{

options.Value1 = 42;

options.Value2 = true;

});

}

````

\* In this example, used the shortcut `Configure<...>` method.

### Get the Option Value

Whenever you need to get the value of an option, [inject](Dependency-Injection.md) the `IOptions<TOption>` service into your class and use its `.Value` property. Example:

````csharp

public class MyService : ITransientDependency

{

private readonly MyOptions \_options;

public MyService(IOptions<MyOptions> options)

{

\_options = options.Value; //Notice the options.Value usage!

}

public void DoIt()

{

var v1 = \_options.Value1;

var v2 = \_options.Value2;

}

}

````

Read [the Microsoft documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options) for all details of the options pattern.

## Pre Configure

One restriction of the options pattern is that you can only resolve (inject) the `IOptions<MyOptions>` and get the option values when the dependency injection configuration completes (that means the `ConfigureServices` methods of all modules complete).

If you are developing a module, you may need to allow developers to set some options and use these options in the dependency injection registration phase. You may need to configure other services or change the dependency injection registration code based on these option values.

For such cases, ABP introduces the `PreConfigure<TOptions>` and the `ExecutePreConfiguredActions<TOptions>` extension methods for the `IServiceCollection`. The pattern works as explained below.

Define a pre option class in your module. Example:

````csharp

public class MyPreOptions

{

public bool MyValue { get; set; }

}

````

Then any [module class](Module-Development-Basics.md) depends on your module can use the `PreConfigure<TOptions>` method in its `PreConfigureServices` method. Example:

````csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<MyPreOptions>(options =>

{

options.MyValue = true;

});

}

````

> Multiple modules can pre-configure the options and override the option values based on their dependency order.

Finally, your module can execute the `ExecutePreConfiguredActions` method in its `ConfigureServices` method to get the configured option values. Example:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var options = context.Services.ExecutePreConfiguredActions<MyPreOptions>();

if (options.MyValue)

{

//...

}

}

````

## Settings

# Settings

[Configuration system](Configuration.md) is a good way to configure the application on startup. In addition to the configurations, ABP provides another way to set and get some application settings.

A setting is a name-value pair stored in a dynamic data source, generally in a database. Setting system is extensible and there are pre-built providers for a user, a tenant, global and default.

## Defining Settings

A setting must be defined before its use. ABP was designed to be [modular](Module-Development-Basics.md), so different modules can have different settings. A module must create a class derived from the `SettingDefinitionProvider` in order to define its settings. An example setting definition provider is shown below:

````csharp

public class EmailSettingProvider : SettingDefinitionProvider

{

public override void Define(ISettingDefinitionContext context)

{

context.Add(

new SettingDefinition("Smtp.Host", "127.0.0.1"),

new SettingDefinition("Smtp.Port", "25"),

new SettingDefinition("Smtp.UserName"),

new SettingDefinition("Smtp.Password", isEncrypted: true),

new SettingDefinition("Smtp.EnableSsl", "false")

);

}

}

````

ABP automatically discovers this class and registers the setting definitions.

### SettingDefinition

`SettingDefinition` class has the following properties:

\* **\*\*Name\*\***: Unique name of the setting in the application. This is **\*\*the only mandatory property\*\***. Used to get/set the value of this setting in the application code (It's a good idea to define a const string for a setting name instead of using a magic string).

\* **\*\*DefaultValue\*\***: A setting may have a default value.

\* **\*\*DisplayName\*\***: A localizable string that can be used to show the setting name on the UI.

\* **\*\*Description\*\***: A localizable string that can be used to show the setting description on the UI.

\* **\*\*IsVisibleToClients\*\***: A boolean value indicates that whether this setting value is available in the client side or not. Default value is false to prevent accidently publishing an internal critical setting value.

\* **\*\*IsInherited\*\***: A boolean value indicates that whether this setting value is inherited from other providers or not. Default value is true and fallbacks to the next provider if the setting value was not set for the requested provider (see the setting value providers section for more).

\* **\*\*IsEncrypted\*\***: A boolean value indicates that whether this setting value should be encrypted on save and decrypted on read. It makes possible to secure the setting value in the database.

\* **\*\*Providers\*\***: Can be used to restrict providers available for a particular setting (see the setting value providers section for more).

\* **\*\*Properties\*\***: A name/value collection to set custom properties about this setting those can be used later in the application code.

### Change Setting Definitions of a Depended Module

In some cases, you may want to change some properties of a settings defined in some other module that your application/module depends on. A setting definition provider can query and update setting definitions.

The following example gets a setting defined by the [Volo.Abp.Emailing](Emailing.md) package and changes its properties:

````csharp

public class MySettingDefinitionProvider : SettingDefinitionProvider

{

public override void Define(ISettingDefinitionContext context)

{

var smtpHost = context.GetOrNull("Abp.Mailing.Smtp.Host");

if (smtpHost != null)

{

smtpHost.DefaultValue = "mail.mydomain.com";

smtpHost.DisplayName =

new LocalizableString(

typeof(MyLocalizationResource),

"SmtpServer\_DisplayName"

);

}

}

}

````

> Using constants for the setting names is a good practice and ABP packages do it. `Abp.Mailing.Smtp.Host` setting name is a constant defined by the `EmailSettingNames` class (in the `Volo.Abp.Emailing` namespace).

## Reading the Setting Values

### ISettingProvider

`ISettingProvider` is used to get the value of a setting or get the values of all the settings. Example usages:

````csharp

public class MyService

{

private readonly ISettingProvider \_settingProvider;

//Inject ISettingProvider in the constructor

public MyService(ISettingProvider settingProvider)

{

\_settingProvider = settingProvider;

}

public async Task FooAsync()

{

//Get a value as string.

string userName = await \_settingProvider.GetOrNullAsync("Smtp.UserName");

//Get a bool value and fallback to the default value (false) if not set.

bool enableSsl = await \_settingProvider.GetAsync<bool>("Smtp.EnableSsl");

//Get a bool value and fallback to the provided default value (true) if not set.

bool enableSsl = await \_settingProvider.GetAsync<bool>(

"Smtp.EnableSsl", defaultValue: true);

//Get a bool value with the IsTrueAsync shortcut extension method

bool enableSsl = await \_settingProvider.IsTrueAsync("Smtp.EnableSsl");

//Get an int value or the default value (0) if not set

int port = (await \_settingProvider.GetAsync<int>("Smtp.Port"));

//Get an int value or null if not provided

int? port = (await \_settingProvider.GetOrNullAsync("Smtp.Port"))?.To<int>();

}

}

````

> `ISettingProvider` is a very common service and some base classes (like `IApplicationService`) already property-inject it. You can directly use the `SettingProvider` property in such cases.

### Reading Setting Values on the Client Side

If a setting is allowed to be visible on the client side, current value of the setting can also be read from the client code. See the following documents to understand how to get the setting values in different UI types;

\* [MVC / Razor Pages](UI/AspNetCore/JavaScript-API/Settings.md)

\* [Angular](UI/Angular/Settings.md)

\* [Blazor](UI/Blazor/Settings.md)

## Setting Value Providers

Setting system is extensible, you can extend it by defining setting value providers to get setting values from any source and based on any condition.

`ISettingProvider` uses the setting value providers to obtain a setting value. It fallbacks to the next value provider if a value provider can not get the setting value.

There are 5 pre-built setting value providers registered by the order below:

\* `DefaultValueSettingValueProvider`: Gets the value from the default value of the setting definition, if set (see the SettingDefinition section above).

\* `ConfigurationSettingValueProvider`: Gets the value from the [IConfiguration service](Configuration.md).

\* `GlobalSettingValueProvider`: Gets the global (system-wide) value for a setting, if set.

\* `TenantSettingValueProvider`: Gets the setting value for the current tenant, if set (see the [multi-tenancy](Multi-Tenancy.md) document).

\* `UserSettingValueProvider`: Gets the setting value for the current user, if set (see the [current user](CurrentUser.md) document).

> Setting fallback system works from bottom (user) to top (default).

Global, Tenant and User setting value providers use the `ISettingStore` to read the value from the data source (see the section below).

### Setting Values in the Application Configuration

As mentioned in the previous section, `ConfigurationSettingValueProvider` reads the settings from the `IConfiguration` service, which can read values from the `appsettings.json` by default. So, the easiest way to configure setting values to define them in the `appsettings.json` file.

For example, you can configure [IEmailSender](Emailing.md) settings as shown below:

````json

{

"Settings": {

"Abp.Mailing.DefaultFromAddress": "noreply@mydomain.com",

"Abp.Mailing.DefaultFromDisplayName": "My Application",

"Abp.Mailing.Smtp.Host": "mail.mydomain.com",

"Abp.Mailing.Smtp.Port": "547",

"Abp.Mailing.Smtp.UserName": "myusername",

"Abp.Mailing.Smtp.Password": "mySecretPassW00rd",

"Abp.Mailing.Smtp.EnableSsl": "True"

}

}

````

Setting values should be configured under the `Settings` section as like in this example.

> `IConfiguration` is an .NET Core service and it can read values not only from the `appsettings.json`, but also from the environment, user secrets... etc. See [Microsoft's documentation]( https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/ ) for more.

### Custom Setting Value Providers

If you need to extend the setting system, you can define a class derived from the `SettingValueProvider` class. Example:

````csharp

public class CustomSettingValueProvider : SettingValueProvider

{

public override string Name => "Custom";

public CustomSettingValueProvider(ISettingStore settingStore)

: base(settingStore)

{

}

public override Task<string> GetOrNullAsync(SettingDefinition setting)

{

/\* Return the setting value or null

Use the SettingStore or another data source \*/

}

}

````

> Alternatively, you can implement the `ISettingValueProvider` interface. Remember to register it to the [dependency injection](Dependency-Injection.md) in this case.

Every provider should have a unique Name (which is "Custom" here). Built-in providers use the given names:

\* `DefaultValueSettingValueProvider`: "**\*\*D\*\***".

\* `ConfigurationSettingValueProvider`: "**\*\*C\*\***".

\* `GlobalSettingValueProvider`: "**\*\*G\*\***".

\* `TenantSettingValueProvider`: "**\*\*T\*\***".

\* `UserSettingValueProvider`: "**\*\*U\*\***".

One-letter names were preferred to reduce the data size in the database (provider name is repeated in each row).

Once you define a custom setting value provider, you need to explicitly register it to the `AbpSettingOptions`:

````csharp

Configure<AbpSettingOptions>(options =>

{

options.ValueProviders.Add<CustomSettingValueProvider>();

});

````

This example adds it as the last item, so it will be the first value provider used by the `ISettingProvider`. You could add it to another index in the `options.ValueProviders` list.

### ISettingStore

While a setting value provider is free to use any source to get the setting value, the `ISettingStore` service is the default source of the setting values. Global, Tenant and User setting value providers use it.

## ISettingEncryptionService

`ISettingEncryptionService` is used to encrypt/decrypt setting values when `IsEncrypted` property of a setting definition was set to `true`.

You can replace this service in the dependency injection system to customize the encryption/decryption process. Default implementation uses the `StringEncryptionService` which is implemented with the AES algorithm by default (see string [encryption document](String-Encryption.md) for more).

## Setting Management Module

The core setting system is pretty independent and doesn't make any assumption about how you manage (change) the setting values. Even the default `ISettingStore` implementation is the `NullSettingStore` which returns null for all setting values.

The setting management module completes it (and implements `ISettingStore`) by managing setting values in a database. See the [Setting Management Module document](Modules/Setting-Management.md) for more.

## Validation

# Validation

Validation system is used to validate the user input or client request for a particular controller action or service method.

ABP is compatible with the ASP.NET Core Model Validation system and everything written in [its documentation](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation) is already valid for ABP based applications. So, this document mostly focuses on the ABP features rather than repeating the Microsoft documentation.

In addition, ABP adds the following benefits:

\* Defines `IValidationEnabled` to add automatic validation to an arbitrary class. Since all the [application services](Application-Services.md) inherently implements it, they are also validated automatically.

\* Automatically localize the validation errors for the data annotation attributes.

\* Provides extensible services to validate a method call or an object state.

\* Provides [FluentValidation](https://fluentvalidation.net/) integration.

## Validating DTOs

This section briefly introduces the validation system. For details, see the [ASP.NET Core validation documentation](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation).

### Data Annotation Attributes

Using data annotations is a simple way to implement the formal validation for a [DTO](Data-Transfer-Objects.md) in a declarative way. Example:

````csharp

public class CreateBookDto

{

[Required]

[StringLength(100)]

public string Name { get; set; }

[Required]

[StringLength(1000)]

public string Description { get; set; }

[Range(0, 999.99)]

public decimal Price { get; set; }

}

````

When you use this class as a parameter to an [application service](Application-Services.md) or a controller, it is automatically validated and a localized validation exception is thrown ([and handled](Exception-Handling.md) by the ABP framework).

### IValidatableObject

`IValidatableObject` can be implemented by a DTO to perform custom validation logic. `CreateBookDto` in the following example implements this interface and checks if the `Name` is equals to the `Description` and returns a validation error in this case.

````csharp

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore

{

public class CreateBookDto : IValidatableObject

{

[Required]

[StringLength(100)]

public string Name { get; set; }

[Required]

[StringLength(1000)]

public string Description { get; set; }

[Range(0, 999.99)]

public decimal Price { get; set; }

public IEnumerable<ValidationResult> Validate(

ValidationContext validationContext)

{

if (Name == Description)

{

yield return new ValidationResult(

"Name and Description can not be the same!",

new[] { "Name", "Description" }

);

}

}

}

}

````

#### Resolving a Service

If you need to resolve a service from the [dependency injection system](Dependency-Injection.md), you can use the `ValidationContext` object. Example:

````csharp

var myService = validationContext.GetRequiredService<IMyService>();

````

> While resolving services in the `Validate` method allows any possibility, it is not a good practice to implement your domain validation logic in DTOs. Keep DTOs simple. Their purpose is to transfer data (DTO: Data Transfer Object).

## Validation Infrastructure

This section explains a few additional services provided by the ABP framework.

### IValidationEnabled Interface

`IValidationEnabled` is an empty marker interface that can be implemented by any class (registered to and resolved from the [DI](Dependency-Injection.md)) to let the ABP framework perform the validation system for the methods of the class. Example:

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Validation;

namespace Acme.BookStore

{

public class MyService : ITransientDependency, IValidationEnabled

{

public virtual async Task DoItAsync(MyInput input)

{

//...

}

}

}

````

> ABP framework uses the [dynamic proxying / interception](Dynamic-Proxying-Interceptors.md) system to perform the validation. In order to make it working, your method should be **\*\*virtual\*\*** or your service should be injected and used over an **\*\*interface\*\*** (like `IMyService`).

#### Enabling/Disabling Validation

You can use the `[DisableValidation]` to disable it for methods, classs and properties.

````csharp

[DisableValidation]

public Void MyMethod()

{

}

[DisableValidation]

public class InputClass

{

public string MyProperty { get; set; }

}

public class InputClass

{

[DisableValidation]

public string MyProperty { get; set; }

}

````

### AbpValidationException

Once ABP determines a validation error, it throws an exception of type `AbpValidationException`. Your application code can throw `AbpValidationException`, but most of the times it is not needed.

\* `ValidationErrors` property of the `AbpValidationException` contains the validation error list.

\* Log level of the `AbpValidationException` is set to `Warning`. It logs all the validation errors to the [logging system](Logging.md).

\* `AbpValidationException` is automatically caught by the ABP framework and converted to a usable error into with HTTP 400 status code. See the [exception handling](Exception-Handling.md) document for more.

## Advanced Topics

### IObjectValidator

In addition to the automatic validation, you may want to manually validate an object. In this case, [inject](Dependency-Injection.md) and use the `IObjectValidator` service:

\* `ValidateAsync` method validates the given object based on the validation rules and throws an `AbpValidationException` if it is not in a valid state.

\* `GetErrorsAsync` doesn't throw an exception, but only returns the validation errors.

`IObjectValidator` is implemented by the `ObjectValidator` by default. `ObjectValidator` is extensible; you can implement `IObjectValidationContributor` interface to contribute a custom logic. Example:

````csharp

public class MyObjectValidationContributor

: IObjectValidationContributor, ITransientDependency

{

public Task AddErrorsAsync(ObjectValidationContext context)

{

//Get the validating object

var obj = context.ValidatingObject;

//Add the validation errors if available

context.Errors.Add(...);

return Task.CompletedTask;

}

}

````

\* Remember to register your class to the [DI](Dependency-Injection.md) (implementing `ITransientDependency` does it just like in this example)

\* ABP will automatically discover your class and use on any type of object validation (including automatic method call validation).

### IMethodInvocationValidator

`IMethodInvocationValidator` is used to validate a method call. It internally uses the `IObjectValidator` to validate objects passes to the method call. You normally don't need to this service since it is automatically used by the framework, but you may want to reuse or replace it on your application in rare cases.

## FluentValidation Integration

Volo.Abp.FluentValidation package integrates the FluentValidation library to the validation system (by implementing the `IObjectValidationContributor`). See the [FluentValidation Integration document](FluentValidation.md) for more.

### FluentValidation Integration

# FluentValidation Integration

ABP [Validation](Validation.md) infrastructure is extensible. [Volo.Abp.FluentValidation](https://www.nuget.org/packages/Volo.Abp.FluentValidation) NuGet package extends the validation system to work with the [FluentValidation](https://fluentvalidation.net/) library.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.FluentValidation

````

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.FluentValidation](https://www.nuget.org/packages/Volo.Abp.FluentValidation) NuGet package to your project:

````

Install-Package Volo.Abp.FluentValidation

````

2. Add the `AbpFluentValidationModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpFluentValidationModule) //Add the FluentValidation module

)]

public class YourModule : AbpModule

{

}

````

## Using the FluentValidation

Follow [the FluentValidation documentation](https://fluentvalidation.net/) to create validator classes. Example:

````csharp

public class CreateUpdateBookDtoValidator : AbstractValidator<CreateUpdateBookDto>

{

public CreateUpdateBookDtoValidator()

{

RuleFor(x => x.Name).Length(3, 10);

RuleFor(x => x.Price).ExclusiveBetween(0.0f, 999.0f);

}

}

````

ABP will automatically find this class and associate with the `CreateUpdateBookDto` on object validation.

## See Also

\* [Validation System](Validation.md)

# Infrastructure

## Audit Logging

# Audit Logging

[Wikipedia](https://en.wikipedia.org/wiki/Audit\_trail): "*\*An audit trail (also called* ***\*\*audit log\*\*****) is a security-relevant chronological record, set of records, and/or destination and source of records that provide documentary evidence of the sequence of activities that have affected at any time a specific operation, procedure, or event\**".

ABP Framework provides an **\*\*extensible audit logging system\*\*** that automates the audit logging by **\*\*convention\*\*** and provides **\*\*configuration\*\*** points to control the level of the audit logs.

An **\*\*audit log object\*\*** (see the Audit Log Object section below) is typically created & saved per web request. It includes;

\* **\*\*Request & response details\*\*** (like URL, Http method, Browser info, HTTP status code... etc.).

\* **\*\*Performed actions\*\*** (controller actions and application service method calls with their parameters).

\* **\*\*Entity changes\*\*** occurred in the web request.

\* **\*\*Exception\*\*** information (if there was an error while executing the request).

\* **\*\*Request duration\*\*** (to measure the performance of the application).

> [Startup templates](Startup-Templates/Index.md) are configured for the audit logging system which is suitable for most of the applications. Use this document for a detailed control over the audit log system.

### Database Provider Support

\* Fully supported by the [Entity Framework Core](Entity-Framework-Core.md) provider.

\* Entity change logging is not supported by the [MongoDB](MongoDB.md) provider. Other features work as expected.

## UseAuditing()

`UseAuditing()` middleware should be added to the ASP.NET Core request pipeline in order to create and save the audit logs. If you've created your applications using [the startup templates](Startup-Templates/Index.md), it is already added.

## AbpAuditingOptions

`AbpAuditingOptions` is the main [options object](Options.md) to configure the audit log system. You can configure it in the `ConfigureServices` method of your [module](Module-Development-Basics.md):

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.IsEnabled = false; //Disables the auditing system

});

````

Here, a list of the options you can configure:

\* `IsEnabled` (default: `true`): A root switch to enable or disable the auditing system. Other options is not used if this value is `false`.

\* `HideErrors` (default: `true`): Audit log system hides and write regular [logs](Logging.md) if any error occurs while saving the audit log objects. If saving the audit logs is critical for your system, set this to `false` to throw exception in case of hiding the errors.

\* `IsEnabledForAnonymousUsers` (default: `true`): If you want to write audit logs only for the authenticated users, set this to `false`. If you save audit logs for anonymous users, you will see `null` for `UserId` values for these users.

\* `AlwaysLogOnException` (default: `true`): If you set to true, it always saves the audit log on an exception/error case without checking other options (except `IsEnabled`, which completely disables the audit logging).

\* `IsEnabledForIntegrationService` (default: `false`): Audit Logging is disabled for [integration services](Integration-Services.md) by default. Set this property as `true` to enable it.

\* `IsEnabledForGetRequests` (default: `false`): HTTP GET requests should not make any change in the database normally and audit log system doesn't save audit log objects for GET request. Set this to `true` to enable it also for the GET requests.

\* `DisableLogActionInfo` (default: `false`):If you set to true, Will no longer log `AuditLogActionInfo`.

\* `ApplicationName`: If multiple applications are saving audit logs into a single database, set this property to your application name, so you can distinguish the logs of different applications. If you don't set, it will set from the `IApplicationInfoAccessor.ApplicationName` value, which is the entry assembly name by default.

\* `IgnoredTypes`: A list of `Type`s to be ignored for audit logging. If this is an entity type, changes for this type of entities will not be saved. This list is also used while serializing the action parameters.

\* `EntityHistorySelectors`: A list of selectors those are used to determine if an entity type is selected for saving the entity change. See the section below for details.

\* `Contributors`: A list of `AuditLogContributor` implementations. A contributor is a way of extending the audit log system. See the "Audit Log Contributors" section below.

\* `AlwaysLogSelectors`: A list of selectors to save the audit logs for the matched criteria.

### Entity History Selectors

Saving all changes of all your entities would require a lot of database space. For this reason, **\*\*audit log system doesn't save any change for the entities unless you explicitly configure it\*\***.

To save all changes of all entities, simply use the `AddAllEntities()` extension method.

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.EntityHistorySelectors.AddAllEntities();

});

````

`options.EntityHistorySelectors` actually a list of type predicate. You can write a lambda expression to define your filter.

The example selector below does the same of the `AddAllEntities()` extension method defined above:

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.EntityHistorySelectors.Add(

new NamedTypeSelector(

"MySelectorName",

type =>

{

if (typeof(IEntity).IsAssignableFrom(type))

{

return true;

}

else

{

return false;

}

}

)

);

});

````

The condition `typeof(IEntity).IsAssignableFrom(type)` will be `true` for any class implements the `IEntity` interface (this is technically all the entities in your application). You can conditionally check and return `true` or `false` based on your preference.

`options.EntityHistorySelectors` is a flexible and dynamic way of selecting the entities for audit logging. Another way is to use the `Audited` and `DisableAuditing` attributes per entity.

## AbpAspNetCoreAuditingOptions

`AbpAspNetCoreAuditingOptions` is the [options object](Options.md) to configure audit logging in the ASP.NET Core layer. You can configure it in the `ConfigureServices` method of your [module](Module-Development-Basics.md):

````csharp

Configure<AbpAspNetCoreAuditingOptions>(options =>

{

options.IgnoredUrls.Add("/products");

});

````

`IgnoredUrls` is the only option. It is a list of ignored URLs prefixes. In the preceding example, all URLs starting with `/products` will be ignored for audit logging.

## Enabling/Disabling Audit Logging for Services

### Enable/Disable for Controllers & Actions

All the controller actions are logged by default (see `IsEnabledForGetRequests` above for GET requests).

You can use the `[DisableAuditing]` to disable it for a specific controller type:

````csharp

[DisableAuditing]

public class HomeController : AbpController

{

//...

}

````

Use `[DisableAuditing]` for any action to control it in the action level:

````csharp

public class HomeController : AbpController

{

[DisableAuditing]

public async Task<ActionResult> Home()

{

//...

}

public async Task<ActionResult> OtherActionLogged()

{

//...

}

}

````

### Enable/Disable for Application Services & Methods

[Application service](Application-Services.md) method calls also included into the audit log by default. You can use the `[DisableAuditing]` in service or method level.

#### Enable/Disable for Other Services

Action audit logging can be enabled for any type of class (registered to and resolved from the [dependency injection](Dependency-Injection.md)) while it is only enabled for the controllers and the application services by default.

Use `[Audited]` and `[DisableAuditing]` for any class or method that need to be audit logged. In addition, your class can (directly or inherently) implement the `IAuditingEnabled` interface to enable the audit logging for that class by default.

### Enable/Disable for Entities & Properties

An entity is ignored on entity change audit logging in the following cases;

\* If you add an entity type to the `AbpAuditingOptions.IgnoredTypes` (as explained before), it is completely ignored in the audit logging system.

\* If the object is not an [entity](Entities.md) (not implements `IEntity` directly or inherently - All entities implement this interface by default).

\* If entity type is not public.

Otherwise, you can use `Audited` to enable entity change audit logging for an entity:

````csharp

[Audited]

public class MyEntity : Entity<Guid>

{

//...

}

````

Or disable it for an entity:

````csharp

[DisableAuditing]

public class MyEntity : Entity<Guid>

{

//...

}

````

Disabling audit logging can be necessary only if the entity is being selected by the `AbpAuditingOptions.EntityHistorySelectors` that explained before.

You can disable auditing only some properties of your entities for a detailed control over the audit logging:

````csharp

[Audited]

public class MyUser : Entity<Guid>

{

public string Name { get; set; }

public string Email { get; set; }

[DisableAuditing] //Ignore the Passoword on audit logging

public string Password { get; set; }

}

````

Audit log system will save changes for the `MyUser` entity while it ignores the `Password` property which can be dangerous to save for security purposes.

In some cases, you may want to save a few properties but ignore all others. Writing `[DisableAuditing]` for all the other properties would be tedious. In such cases, use `[Audited]` only for the desired properties and mark the entity with the `[DisableAuditing]` attribute:

````csharp

[DisableAuditing]

public class MyUser : Entity<Guid>

{

[Audited] //Only log the Name change

public string Name { get; set; }

public string Email { get; set; }

public string Password { get; set; }

}

````

## IAuditingStore

`IAuditingStore` is an interface that is used to save the audit log objects (explained below) by the ABP Framework. If you need to save the audit log objects to a custom data store, you can implement the `IAuditingStore` in your own application and replace using the [dependency injection system](Dependency-Injection.md).

`SimpleLogAuditingStore` is used if no audit store was registered. It simply writes the audit object to the standard [logging system](Logging.md).

[The Audit Logging Module](Modules/Audit-Logging.md) has been configured in [the startup templates](Startup-Templates/Index.md) saves audit log objects to a database (it supports multiple database providers). So, most of the times you don't care about how `IAuditingStore` was implemented and used.

## Audit Log Object

An **\*\*audit log object\*\*** is created for each **\*\*web request\*\*** by default. An audit log object can be represented by the following relation diagram:

![\*\*auditlog-object-diagram\*\*](images/auditlog-object-diagram.png)

\* **\*\*AuditLogInfo\*\***: The root object with the following properties:

\* `ApplicationName`: When you save audit logs of different applications to the same database, this property is used to distinguish the logs of the applications.

\* `UserId`: Id of the current user, if the user has logged in.

\* `UserName`: User name of the current user, if the user has logged in (this value is here to not depend on the identity module/system for lookup).

\* `TenantId`: Id of the current tenant, for a multi-tenant application.

\* `TenantName`: Name of the current tenant, for a multi-tenant application.

\* `ExecutionTime`: The time when this audit log object has been created.

\* `ExecutionDuration`: Total execution duration of the request, in milliseconds. This can be used to observe the performance of the application.

\* `ClientId`: Id of the current client, if the client has been authenticated. A client is generally a 3rd-party application using the system over an HTTP API.

\* `ClientName`: Name of the current client, if available.

\* `ClientIpAddress`: IP address of the client/user device.

\* `CorrelationId`: Current [Correlation Id](CorrelationId.md). Correlation Id is used to relate the audit logs written by different applications (or microservices) in a single logical operation.

\* `BrowserInfo`: Browser name/version info of the current user, if available.

\* `HttpMethod`: HTTP method of the current request (GET, POST, PUT, DELETE... etc.).

\* `HttpStatusCode`: HTTP response status code for this request.

\* `Url`: URL of the request.

\* **\*\*AuditLogActionInfo\*\***: An audit log action is typically a controller action or an [application service](Application-Services.md) method call during the web request. One audit log may contain multiple actions. An action object has the following properties:

\* `ServiceName`: Name of the executed controller/service.

\* `MethodName`: Name of the executed method of the controller/service.

\* `Parameters`: A JSON formatted text representing the parameters passed to the method.

\* `ExecutionTime`: The time when this method was executed.

\* `ExecutionDuration`: Duration of the method execution, in milliseconds. This can be used to observe the performance of the method.

\* **\*\*EntityChangeInfo\*\***: Represents a change of an entity in this web request. An audit log may contain zero or more entity changes. An entity change has the following properties:

\* `ChangeTime`: The time when the entity was changed.

\* `ChangeType`: An enum with the following fields: `Created` (0), `Updated` (1) and `Deleted` (2).

\* `EntityId`: Id of the entity that was changed.

\* `EntityTenantId`: Id of the tenant this entity belongs to.

\* `EntityTypeFullName`: Type (class) name of the entity with full namespace (like *\*Acme.BookStore.Book\** for the Book entity).

\* **\*\*EntityPropertyChangeInfo\*\***: Represents a change of a property of an entity. An entity change info (explained above) may contain one or more property change with the following properties:

\* `NewValue`: New value of the property. It is `null` if the entity was deleted.

\* `OriginalValue`: Old/original value before the change. It is `null` if the entity was newly created.

\* `PropertyName`: The name of the property on the entity class.

\* `PropertyTypeFullName`: Type (class) name of the property with full namespace.

\* **\*\*Exception\*\***: An audit log object may contain zero or more exception. In this way, you can get a report of the failed requests.

\* **\*\*Comment\*\***: An arbitrary string value to add custom messages to the audit log entry. An audit log object may contain zero or more comments.

In addition to the standard properties explained above, `AuditLogInfo`, `AuditLogActionInfo` and `EntityChangeInfo` objects implement the `IHasExtraProperties` interface, so you can add custom properties to these objects.

## Audit Log Contributors

You can extend the auditing system by creating a class that is derived from the `AuditLogContributor` class which defines the `PreContribute` and the `PostContribute` methods.

The only pre-built contributor is the `AspNetCoreAuditLogContributor` class which sets the related properties for an HTTP request.

A contributor can set properties and collections of the `AuditLogInfo` class to add more information.

Example:

````csharp

public class MyAuditLogContributor : AuditLogContributor

{

public override void PreContribute(AuditLogContributionContext context)

{

var currentUser = context.ServiceProvider.GetRequiredService<ICurrentUser>();

context.AuditInfo.SetProperty(

"MyCustomClaimValue",

currentUser.FindClaimValue("MyCustomClaim")

);

}

public override void PostContribute(AuditLogContributionContext context)

{

context.AuditInfo.Comments.Add("Some comment...");

}

}

````

\* `context.ServiceProvider` can be used to resolve services from the [dependency injection](Dependency-Injection.md).

\* `context.AuditInfo` can be used to access to the current audit log object to manipulate it.

After creating such a contributor, you must add it to the `AbpAuditingOptions.Contributors` list:

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.Contributors.Add(new MyAuditLogContributor());

});

````

## IAuditLogScope & IAuditingManager

This section explains the `IAuditLogScope` & `IAuditingManager` services for advanced use cases.

An **\*\*audit log scope\*\*** is an [ambient scope](Ambient-Context-Pattern.md) that **\*\*builds\*\*** and **\*\*saves\*\*** an audit log object (explained before). By default, an audit log scope is created for a web request by the Audit Log Middleware (see `UseAuditing()` section above).

### Access to the Current Audit Log Scope

Audit log contributors, was explained above, is a global way of manipulating the audit log object. It is good if you can get a value from a service.

If you need to manipulate the audit log object in an arbitrary point of your application, you can access to the current audit log scope and get the current audit log object (independent of how the scope is managed). Example:

````csharp

public class MyService : ITransientDependency

{

private readonly IAuditingManager \_auditingManager;

public MyService(IAuditingManager auditingManager)

{

\_auditingManager = auditingManager;

}

public async Task DoItAsync()

{

var currentAuditLogScope = \_auditingManager.Current;

if (currentAuditLogScope != null)

{

currentAuditLogScope.Log.Comments.Add(

"Executed the MyService.DoItAsync method :)"

);

currentAuditLogScope.Log.SetProperty("MyCustomProperty", 42);

}

}

}

````

Always check if `\_auditingManager.Current` is null or not, because it is controlled in an outer scope and you can't know if an audit log scope was created before calling your method.

### Manually Create an Audit Log Scope

You rarely need to create a manual audit log scope, but if you need, you can create an audit log scope using the `IAuditingManager` as like in the following example:

````csharp

public class MyService : ITransientDependency

{

private readonly IAuditingManager \_auditingManager;

public MyService(IAuditingManager auditingManager)

{

\_auditingManager = auditingManager;

}

public async Task DoItAsync()

{

using (var auditingScope = \_auditingManager.BeginScope())

{

try

{

//Call other services...

}

catch (Exception ex)

{

//Add exceptions

\_auditingManager.Current.Log.Exceptions.Add(ex);

throw;

}

finally

{

//Always save the log

await auditingScope.SaveAsync();

}

}

}

}

````

You can call other services, they may call others, they may change entities and so on. All these interactions are saved as a single audit log object in the finally block.

## The Audit Logging Module

The Audit Logging Module basically implements the `IAuditingStore` to save the audit log objects to a database. It supports multiple database providers. This module is added to the startup templates by default.

See [the Audit Logging Module document](Modules/Audit-Logging.md) for more about it.

## Background Jobs

# Background Jobs

## Introduction

Background jobs are used to queue some tasks to be executed in the background. You may need background jobs for several reasons. Here are some examples:

- To perform **\*\*long-running tasks\*\*** without having the users wait. For example, a user presses a 'report' button to start a long-running reporting job. You add this job to the **\*\*queue\*\*** and send the report's result to your user via email when it's completed.

- To create **\*\*re-trying\*\*** and **\*\*persistent tasks\*\*** to **\*\*guarantee\*\*** that a code will be **\*\*successfully executed\*\***. For example, you can send emails in a background job to overcome **\*\*temporary failures\*\*** and **\*\*guarantee\*\*** that it eventually will be sent. That way users do not wait while sending emails.

Background jobs are **\*\*persistent\*\*** that means they will be **\*\*re-tried\*\*** and **\*\*executed\*\*** later even if your application crashes.

## Abstraction Package

ABP provides an **\*\*abstraction\*\*** module and **\*\*several implementations\*\*** for background jobs. It has a built-in/default implementation as well as Hangfire, RabbitMQ and Quartz integrations.

`Volo.Abp.BackgroundJobs.Abstractions` NuGet package provides needed services to create background jobs and queue background job items. If your module only depend on this package, it can be independent from the actual implementation/integration.

> `Volo.Abp.BackgroundJobs.Abstractions` package is installed to the startup templates by default.

### Create a Background Job

A background job is a class that implements the `IBackgroundJob<TArgs>` interface or derives from the `BackgroundJob<TArgs>` class. `TArgs` is a simple plain C# class to store the job data.

This example is used to send emails in background. First, define a class to store arguments of the background job:

````csharp

namespace MyProject

{

public class EmailSendingArgs

{

public string EmailAddress { get; set; }

public string Subject { get; set; }

public string Body { get; set; }

}

}

````

Then create a background job class that uses an `EmailSendingArgs` object to send an email:

````csharp

using System.Threading.Tasks;

using Volo.Abp.BackgroundJobs;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

namespace MyProject

{

public class EmailSendingJob

: AsyncBackgroundJob<EmailSendingArgs>, ITransientDependency

{

private readonly IEmailSender \_emailSender;

public EmailSendingJob(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public override async Task ExecuteAsync(EmailSendingArgs args)

{

await \_emailSender.SendAsync(

args.EmailAddress,

args.Subject,

args.Body

);

}

}

}

````

This job simply uses `IEmailSender` to send emails (see [email sending document](Emailing.md)).

> `AsyncBackgroundJob` is used to create a job needs to perform async calls. You can inherit from `BackgroundJob<TJob>` and override the `Execute` method if the method doesn't need to perform any async call.

#### Exception Handling

A background job should not hide exceptions. If it throws an exception, the background job is automatically re-tried after a calculated waiting time. Hide exceptions only if you don't want to re-run the background job for the current argument.

#### Cancelling Background Jobs

If your background task is cancellable, then you can use the standard [Cancellation Token](Cancellation-Token-Provider.md) system to obtain a `CancellationToken` to cancel your job when requested. See the following example that uses the `ICancellationTokenProvider` to obtain the cancellation token:

```csharp

using System;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Threading;

namespace MyProject

{

public class LongRunningJob : AsyncBackgroundJob<LongRunningJobArgs>, ITransientDependency

{

private readonly ICancellationTokenProvider \_cancellationTokenProvider;

public LongRunningJob(ICancellationTokenProvider cancellationTokenProvider)

{

\_cancellationTokenProvider = cancellationTokenProvider;

}

public override async Task ExecuteAsync(LongRunningJobArgs args)

{

foreach (var id in args.Ids)

{

\_cancellationTokenProvider.Token.ThrowIfCancellationRequested();

await ProcessAsync(id); // code omitted for brevity

}

}

}

}

```

> A cancellation operation might be needed if the application is shutting down and we don't want to block the application in the background job. This example throws an exception if the cancellation is requested. So, the job will be retried the next time the application starts. If you don't want that, just return from the `ExecuteAsync` method without throwing any exception (you can simply check the `\_cancellationTokenProvider.Token.IsCancellationRequested` property).

#### Job Name

Each background job has a name. Job names are used in several places. For example, RabbitMQ provider uses job names to determine the RabbitMQ Queue names.

Job name is determined by the **\*\*job argument type\*\***. For the `EmailSendingArgs` example above, the job name is `MyProject.EmailSendingArgs` (full name, including the namespace). You can use the `BackgroundJobName` attribute to set a different job name.

**\*\*Example\*\***

```csharp

using Volo.Abp.BackgroundJobs;

namespace MyProject

{

[BackgroundJobName("emails")]

public class EmailSendingArgs

{

public string EmailAddress { get; set; }

public string Subject { get; set; }

public string Body { get; set; }

}

}

```

### Queue a Job Item

Now, you can queue an email sending job using the `IBackgroundJobManager` service:

````csharp

public class RegistrationService : ApplicationService

{

private readonly IBackgroundJobManager \_backgroundJobManager;

public RegistrationService(IBackgroundJobManager backgroundJobManager)

{

\_backgroundJobManager = backgroundJobManager;

}

public async Task RegisterAsync(string userName, string emailAddress, string password)

{

//TODO: Create new user in the database...

await \_backgroundJobManager.EnqueueAsync(

new EmailSendingArgs

{

EmailAddress = emailAddress,

Subject = "You've successfully registered!",

Body = "..."

}

);

}

}

````

Just injected `IBackgroundJobManager` service and used its `EnqueueAsync` method to add a new job to the queue.

Enqueue method gets some optional arguments to control the background job:

\* **\*\*priority\*\*** is used to control priority of the job item. It gets an `BackgroundJobPriority` enum which has `Low`, `BelowNormal`, `Normal` (default), `AboveNormal` and `Hight` fields.

\* **\*\*delay\*\*** is used to wait a while (`TimeSpan`) before first try.

### Disable Job Execution

You may want to disable background job execution for your application. This is generally needed if you want to execute background jobs in another process and disable it for the current process.

Use `AbpBackgroundJobOptions` to configure the job execution:

````csharp

[DependsOn(typeof(AbpBackgroundJobsModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobOptions>(options =>

{

options.IsJobExecutionEnabled = false; //Disables job execution

});

}

}

````

## Default Background Job Manager

ABP framework includes a simple `IBackgroundJobManager` implementation that;

- Works as **\*\*FIFO\*\*** in a **\*\*single thread\*\***.

- **\*\*Retries\*\*** job execution until the job **\*\*successfully runs\*\*** or **\*\*timeouts\*\***. Default timeout is 2 days for a job. Logs all exceptions.

- **\*\*Deletes\*\*** a job from the store (database) when it's successfully executed. If it's timed out, it sets it as **\*\*abandoned\*\*** and leaves it in the database.

- **\*\*Increasingly waits between retries\*\*** for a job. It waits 1 minute for the first retry, 2 minutes for the second retry, 4 minutes for the third retry and so on.

- **\*\*Polls\*\*** the store for jobs in fixed intervals. It queries jobs, ordering by priority (asc) and then by try count (asc).

> `Volo.Abp.BackgroundJobs` nuget package contains the default background job manager and it is installed to the startup templates by default.

### Configuration

Use `AbpBackgroundJobWorkerOptions` in your [module class](Module-Development-Basics.md) to configure the default background job manager. The example below changes the timeout duration for background jobs:

````csharp

[DependsOn(typeof(AbpBackgroundJobsModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobWorkerOptions>(options =>

{

options.DefaultTimeout = 864000; //10 days (as seconds)

});

}

}

````

### Data Store

The default background job manager needs a data store to save and read jobs. It defines `IBackgroundJobStore` as an abstraction to store the jobs.

Background Jobs module implements `IBackgroundJobStore` using various data access providers. See its own [documentation](Modules/Background-Jobs.md). If you don't want to use this module, you should implement the `IBackgroundJobStore` interface yourself.

> Background Jobs module is already installed to the startup templates by default and it works based on your ORM/data access choice.

### Clustered Deployment

The default background job manager is compatible with [clustered environments](Deployment/Clustered-Environment.md) (where multiple instances of your application run concurrently). It uses a [distributed lock](Distributed-Locking.md) to ensure that the jobs are executed only in a single application instance at a time.

However, the distributed lock system works in-process by default. That means it is not distributed actually, unless you configure a distributed lock provider. So, **\*\*please follow the [**distributed lock**](Distributed-Locking.md) document to configure a provider for your application\*\***, if it is not already configured.

If you don't want to use a distributed lock provider, you may go with the following options:

\* Stop the background job manager (set `AbpBackgroundJobOptions.IsJobExecutionEnabled` to `false` as explained in the *\*Disable Job Execution\** section) in all application instances except one of them, so only the single instance executes the jobs (while other application instances can still queue jobs).

\* Stop the background job manager (set `AbpBackgroundJobOptions.IsJobExecutionEnabled` to `false` as explained in the *\*Disable Job Execution\** section) in all application instances and create a dedicated application (maybe a console application running in its own container or a Windows Service running in the background) to execute all the background jobs. This can be a good option if your background jobs consume high system resources (CPU, RAM or Disk), so you can deploy that background application to a dedicated server and your background jobs don't affect your application's performance.

## Integrations

Background job system is extensible and you can change the default background job manager with your own implementation or on of the pre-built integrations.

See pre-built job manager alternatives:

\* [Hangfire Background Job Manager](Background-Jobs-Hangfire.md)

\* [RabbitMQ Background Job Manager](Background-Jobs-RabbitMq.md)

\* [Quartz Background Job Manager](Background-Jobs-Quartz.md)

## See Also

\* [Background Workers](Background-Workers.md)

### Hangfire Integration

# Hangfire Background Job Manager

[Hangfire](https://www.hangfire.io/) is an advanced background job manager. You can integrate Hangfire with the ABP Framework to use it instead of the [default background job manager](Background-Jobs.md). In this way, you can use the same background job API for Hangfire and your code will be independent of Hangfire. If you like, you can directly use Hangfire's API, too.

> See the [background jobs document](Background-Jobs.md) to learn how to use the background job system. This document only shows how to install and configure the Hangfire integration.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.BackgroundJobs.HangFire

````

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.BackgroundJobs.HangFire).

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.BackgroundJobs.HangFire](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.HangFire) NuGet package to your project:

````

Install-Package Volo.Abp.BackgroundJobs.HangFire

````

2. Add the `AbpBackgroundJobsHangfireModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsHangfireModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## Configuration

You can install any storage for Hangfire. The most common one is SQL Server (see the [Hangfire.SqlServer](https://www.nuget.org/packages/Hangfire.SqlServer) NuGet package).

After you have installed these NuGet packages, you need to configure your project to use Hangfire.

1.First, we change the `Module` class (example: `<YourProjectName>HttpApiHostModule`) to add Hangfire configuration of the storage and connection string in the `ConfigureServices` method:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

var hostingEnvironment = context.Services.GetHostingEnvironment();

//... other configurations.

ConfigureHangfire(context, configuration);

}

private void ConfigureHangfire(ServiceConfigurationContext context, IConfiguration configuration)

{

context.Services.AddHangfire(config =>

{

config.UseSqlServerStorage(configuration.GetConnectionString("Default"));

});

}

````

> You have to configure a storage for Hangfire.

2. If you want to use hangfire's dashboard, you can add `UseAbpHangfireDashboard` call in the `OnApplicationInitialization` method in `Module` class:

````csharp

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

// ... others

app.UseAbpHangfireDashboard(); //should add to the request pipeline before the app.UseConfiguredEndpoints()

app.UseConfiguredEndpoints();

}

````

### Specifying Queue

You can use the [`QueueAttribute`](https://docs.hangfire.io/en/latest/background-processing/configuring-queues.html) to specify the queue:

````csharp

using System.Threading.Tasks;

using Volo.Abp.BackgroundJobs;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

namespace MyProject

{

[Queue("alpha")]

public class EmailSendingJob

: AsyncBackgroundJob<EmailSendingArgs>, ITransientDependency

{

private readonly IEmailSender \_emailSender;

public EmailSendingJob(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public override async Task ExecuteAsync(EmailSendingArgs args)

{

await \_emailSender.SendAsync(

args.EmailAddress,

args.Subject,

args.Body

);

}

}

}

````

### Dashboard Authorization

Hangfire Dashboard provides information about your background jobs, including method names and serialized arguments as well as gives you an opportunity to manage them by performing different actions – retry, delete, trigger, etc. So it is important to restrict access to the Dashboard.

To make it secure by default, only local requests are allowed, however you can change this by following the [official documentation](http://docs.hangfire.io/en/latest/configuration/using-dashboard.html) of Hangfire.

You can integrate the Hangfire dashboard to [ABP authorization system](Authorization.md) using the **\*\*AbpHangfireAuthorizationFilter\*\***

class. This class is defined in the `Volo.Abp.Hangfire` package. The following example, checks if the current user is logged in to the application:

```csharp

app.UseAbpHangfireDashboard("/hangfire", options =>

{

options.AsyncAuthorization = new[] { new AbpHangfireAuthorizationFilter() };

});

```

\* `AbpHangfireAuthorizationFilter` is an implementation of an authorization filter.

#### AbpHangfireAuthorizationFilter

`AbpHangfireAuthorizationFilter` class has the following fields:

\* **\*\***`enableTenant` **(**`bool`**, default:** `false`**):\*\*** Enables/disables accessing the Hangfire dashboard on tenant users.

\* **\*\***`requiredPermissionName` **(**`string`**, default:** `null`**):\*\*** Hangfire dashboard is accessible only if the current user has the specified permission. In this case, if we specify a permission name, we don't need to set `enableTenant` `true` because the permission system already does it.

If you want to require an additional permission, you can pass it into the constructor as below:

```csharp

app.UseAbpHangfireDashboard("/hangfire", options =>

{

options.AsyncAuthorization = new[] { new AbpHangfireAuthorizationFilter(requiredPermissionName: "MyHangFireDashboardPermissionName") };

});

```

**\*\*Important\*\***: `UseAbpHangfireDashboard` should be called after the authentication and authorization middlewares in your `Startup` class (probably at the last line). Otherwise,

authorization will always fail!

### RabbitMQ Integration

# RabbitMQ Background Job Manager

RabbitMQ is an industry standard message broker. While it is typically used for inter-process communication (messaging / distributed events), it is pretty useful to store and execute background jobs in FIFO (First In First Out) order.

ABP Framework provides the [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) NuGet package to use the RabbitMQ for background job execution.

> See the [background jobs document](Background-Jobs.md) to learn how to use the background job system. This document only shows how to install and configure the RabbitMQ integration.

## Installation

Use the ABP CLI to add [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BackgroundJobs.RabbitMQ` package.

\* Run `abp add-package Volo.Abp.BackgroundJobs.RabbitMQ` command.

If you want to do it manually, install the [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) NuGet package to your project and add `[DependsOn(typeof(AbpBackgroundJobsRabbitMqModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

### Default Configuration

The default configuration automatically connects to the local RabbitMQ server (localhost) with the standard port. **\*\*In this case, no configuration needed.\*\***

### RabbitMQ Connection(s)

You can configure the RabbitMQ connections using the standard [configuration system](Configuration.md), like using the `appsettings.json` file, or using the [options](Options.md) classes.

#### `appsettings.json` file configuration

This is the simplest way to configure the RabbitMQ connections. It is also very strong since you can use any other configuration source (like environment variables) that is [supported by the AspNet Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/).

**\*\*Example: Configuring the Default RabbitMQ Connection\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123",

"Port": "5672"

}

}

}

}

````

You can use any of the [ConnectionFactry](http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.ConnectionFactory.html#properties) properties as the connection properties. See [the RabbitMQ document](https://www.rabbitmq.com/dotnet-api-guide.html#exchanges-and-queues) to understand these options better.

Defining multiple connections is allowed. In this case, you can use different connections for different background job types (see the `AbpRabbitMqBackgroundJobOptions` section below).

**\*\*Example: Declare two connections\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

},

"SecondConnection": {

"HostName": "321.321.321.321"

}

}

}

}

````

If you need to connect to the RabbitMQ cluster, you can use the `;` character to separate the host names.

**\*\*Example: Connect to the RabbitMQ cluster\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123;234.234.234.234"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

```

#### AbpRabbitMqOptions

`AbpRabbitMqOptions` class can be used to configure the connection strings for the RabbitMQ. You can configure this options inside the `ConfigureServices` of your [module](Module-Development-Basics.md).

**\*\*Example: Configure the connection\*\***

````csharp

Configure<AbpRabbitMqOptions>(options =>

{

options.Connections.Default.UserName = "user";

options.Connections.Default.Password = "pass";

options.Connections.Default.HostName = "123.123.123.123";

options.Connections.Default.Port = 5672;

});

````

Using these options classes can be combined with the `appsettings.json` way. Configuring an option property in the code overrides the value in the configuration file.

### AbpRabbitMqBackgroundJobOptions

#### Job Queue Names

By default, each job type uses a separate queue. Queue names are calculated by combining a standard prefix and the job name. Default prefix is `AbpBackgroundJobs.` So, if the job name is `EmailSending` then the queue name in the RabbitMQ becomes `AbpBackgroundJobs.EmailSending`

> Use `BackgroundJobName` attribute on the background **\*\*job argument\*\*** class to specify the job name. Otherwise, the job name will be the full name (with namespace) of the job class.

#### Job Connections

By default, all the job types use the `Default` RabbitMQ connection.

#### Customization

`AbpRabbitMqBackgroundJobOptions` can be used to customize the queue names and the connections used by the jobs.

**\*\*Example:\*\***

````csharp

Configure<AbpRabbitMqBackgroundJobOptions>(options =>

{

options.DefaultQueueNamePrefix = "my\_app\_jobs.";

options.DefaultDelayedQueueNamePrefix = "my\_app\_jobs.delayed"

options.PrefetchCount = 1;

options.JobQueues[typeof(EmailSendingArgs)] =

new JobQueueConfiguration(

typeof(EmailSendingArgs),

queueName: "my\_app\_jobs.emails",

connectionName: "SecondConnection",

delayedQueueName:"my\_app\_jobs.emails.delayed"

);

});

````

\* This example sets the default queue name prefix to `my\_app\_jobs.` and default delayed queue name prefix to `my\_app\_jobs.delayed`. If different applications use the same RabbitMQ server, it would be important to use different prefixes for each application to not consume jobs of each other.

\* Sets `PrefetchCount` for all queues.

\* Also specifies a different connection string for the `EmailSendingArgs`.

`JobQueueConfiguration` class has some additional options in its constructor;

\* `queueName`: The queue name that is used for this job. The prefix is not added, so you need to specify the full name of the queue.

\* `DelayedQueueName`: The delayed queue name that is used for delayed execution of job. The prefix is not added, so you need to specify the full name of the queue.

\* `connectionName`: The RabbitMQ connection name (see the connection configuration above). This is optional and the default value is `Default`.

\* `durable` (optional, default: `true`).

\* `exclusive` (optional, default: `false`).

\* `autoDelete` (optional, default: `false`).

\* `PrefetchCount` (optional, default: null)

See the RabbitMQ documentation if you want to understand the `durable`, `exclusive` and `autoDelete` options better, while most of the times the default configuration is what you want.

## See Also

\* [Background Jobs](Background-Jobs.md)

### Quartz Integration

# Quartz Background Job Manager

[Quartz](https://www.quartz-scheduler.net/) is an advanced background job manager. You can integrate Quartz with the ABP Framework to use it instead of the [default background job manager](Background-Jobs.md). In this way, you can use the same background job API for Quartz and your code will be independent of Quartz. If you like, you can directly use Quartz's API, too.

> See the [background jobs document](Background-Jobs.md) to learn how to use the background job system. This document only shows how to install and configure the Quartz integration.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.BackgroundJobs.Quartz

````

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.BackgroundJobs.Quartz).

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.BackgroundJobs.Quartz](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.Quartz) NuGet package to your project:

````

Install-Package Volo.Abp.BackgroundJobs.Quartz

````

2. Add the `AbpBackgroundJobsQuartzModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## Configuration

Quartz is a very configurable library,and the ABP framework provides `AbpQuartzOptions` for this. You can use the `PreConfigure` method in your module class to pre-configure this option. ABP will use it when initializing the Quartz module. For example:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

PreConfigure<AbpQuartzOptions>(options =>

{

options.Properties = new NameValueCollection

{

["quartz.jobStore.dataSource"] = "BackgroundJobsDemoApp",

["quartz.jobStore.type"] = "Quartz.Impl.AdoJobStore.JobStoreTX, Quartz",

["quartz.jobStore.tablePrefix"] = "QRTZ\_",

["quartz.serializer.type"] = "json",

["quartz.dataSource.BackgroundJobsDemoApp.connectionString"] = configuration.GetConnectionString("Quartz"),

["quartz.dataSource.BackgroundJobsDemoApp.provider"] = "SqlServer",

["quartz.jobStore.driverDelegateType"] = "Quartz.Impl.AdoJobStore.SqlServerDelegate, Quartz",

};

});

}

}

````

Starting from ABP 3.1 version, we have added `Configurator` to `AbpQuartzOptions` to configure Quartz. For example:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

PreConfigure<AbpQuartzOptions>(options =>

{

options.Configurator = configure =>

{

configure.UsePersistentStore(storeOptions =>

{

storeOptions.UseProperties = true;

storeOptions.UseJsonSerializer();

storeOptions.UseSqlServer(configuration.GetConnectionString("Quartz"));

storeOptions.UseClustering(c =>

{

c.CheckinMisfireThreshold = TimeSpan.FromSeconds(20);

c.CheckinInterval = TimeSpan.FromSeconds(10);

});

});

};

});

}

}

````

> You can choose the way you favorite to configure Quaratz.

Quartz stores job and scheduling information **\*\*in memory by default\*\***. In the example, we use the pre-configuration of [options pattern](Options.md) to change it to the database. For more configuration of Quartz, please refer to the Quartz's [documentation](https://www.quartz-scheduler.net/).

## Exception handling

### Default exception handling strategy

When an exception occurs in the background job,ABP provide the **\*\*default handling strategy\*\*** retrying once every 3 seconds, up to 3 times. You can change the retry count and retry interval via `AbpBackgroundJobQuartzOptions` options:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobQuartzOptions>(options =>

{

options.RetryCount = 1;

options.RetryIntervalMillisecond = 1000;

});

}

}

```

### Customize exception handling strategy

You can customize the exception handling strategy via `AbpBackgroundJobQuartzOptions` options:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobQuartzOptions>(options =>

{

options.RetryStrategy = async (retryIndex, executionContext, exception) =>

{

// customize exception handling

};

});

}

}

```

## Background Workers

# Background Workers

## Introduction

Background workers are simple independent threads in the application running in the background. Generally, they run periodically to perform some tasks. Examples;

\* A background worker can run periodically to **\*\*delete old logs\*\***.

\* A background worker can run periodically to **\*\*determine inactive users\*\*** and **\*\*send emails\*\*** to get users to return to your application.

## Create a Background Worker

A background worker should directly or indirectly implement the `IBackgroundWorker` interface.

> A background worker is inherently [singleton](Dependency-Injection.md). So, only a single instance of your worker class is instantiated and run.

### BackgroundWorkerBase

`BackgroundWorkerBase` is an easy way to create a background worker.

````csharp

public class MyWorker : BackgroundWorkerBase

{

public override Task StartAsync(CancellationToken cancellationToken = default)

{

//...

}

public override Task StopAsync(CancellationToken cancellationToken = default)

{

//...

}

}

````

Start your worker in the `StartAsync` (which is called when the application begins) and stop in the `StopAsync` (which is called when the application shuts down).

> You can directly implement the `IBackgroundWorker`, but `BackgroundWorkerBase` provides some useful properties like `Logger`.

### AsyncPeriodicBackgroundWorkerBase

Assume that we want to make a user passive, if the user has not logged in to the application in last 30 days. `AsyncPeriodicBackgroundWorkerBase` class simplifies to create periodic workers, so we will use it for the example below:

````csharp

public class PassiveUserCheckerWorker : AsyncPeriodicBackgroundWorkerBase

{

public PassiveUserCheckerWorker(

AbpAsyncTimer timer,

IServiceScopeFactory serviceScopeFactory

) : base(

timer,

serviceScopeFactory)

{

Timer.Period = 600000; //10 minutes

}

protected async override Task DoWorkAsync(

PeriodicBackgroundWorkerContext workerContext)

{

Logger.LogInformation("Starting: Setting status of inactive users...");

//Resolve dependencies

var userRepository = workerContext

.ServiceProvider

.GetRequiredService<IUserRepository>();

//Do the work

await userRepository.UpdateInactiveUserStatusesAsync();

Logger.LogInformation("Completed: Setting status of inactive users...");

}

}

````

\* `AsyncPeriodicBackgroundWorkerBase` uses the `AbpTimer` (a thread-safe timer) object to determine **\*\*the period\*\***. We can set its `Period` property in the constructor.

\* It required to implement the `DoWorkAsync` method to **\*\*execute\*\*** the periodic work.

\* It is a good practice to **\*\*resolve dependencies\*\*** from the `PeriodicBackgroundWorkerContext` instead of constructor injection. Because `AsyncPeriodicBackgroundWorkerBase` uses a `IServiceScope` that is **\*\*disposed\*\*** when your work finishes.

\* `AsyncPeriodicBackgroundWorkerBase` **\*\*catches and logs exceptions\*\*** thrown by the `DoWorkAsync` method.

## Register Background Worker

After creating a background worker class, you should add it to the `IBackgroundWorkerManager`. The most common place is the `OnApplicationInitializationAsync` method of your module class:

````csharp

[DependsOn(typeof(AbpBackgroundWorkersModule))]

public class MyModule : AbpModule

{

public override async Task OnApplicationInitializationAsync(

ApplicationInitializationContext context)

{

await context.AddBackgroundWorkerAsync<PassiveUserCheckerWorker>();

}

}

````

`context.AddBackgroundWorkerAsync(...)` is a shortcut extension method for the expression below:

````csharp

await context.ServiceProvider

.GetRequiredService<IBackgroundWorkerManager>()

.AddAsync(

context

.ServiceProvider

.GetRequiredService<PassiveUserCheckerWorker>()

);

````

So, it resolves the given background worker and adds to the `IBackgroundWorkerManager`.

While we generally add workers in `OnApplicationInitializationAsync`, there are no restrictions on that. You can inject `IBackgroundWorkerManager` anywhere and add workers at runtime. Background worker manager will stop and release all the registered workers when your application is being shut down.

## Options

`AbpBackgroundWorkerOptions` class is used to [set options](Options.md) for the background workers. Currently, there is only one option:

\* `IsEnabled` (default: true): Used to **\*\*enable/disable\*\*** the background worker system for your application.

> See the [Options](Options.md) document to learn how to set options.

## Making Your Application Always Run

Background workers only work if your application is running. If you host the background job execution in your web application (this is the default behavior), you should ensure that your web application is configured to always be running. Otherwise, background jobs only work while your application is in use.

## Running On a Cluster

Be careful if you run multiple instances of your application simultaneously in a clustered environment. In that case, every application runs the same worker which may create conflicts if your workers are running on the same resources (processing the same data, for example).

If that's a problem for your workers, you have the following options:

\* Implement your background workers so that they work in a clustered environment without any problem. Using the [distributed lock](Distributed-Locking.md) to ensure concurrency control is a way of doing that. A background worker in an application instance may handle a distributed lock, so the workers in other application instances will wait for the lock. In this way, only one worker does the actual work, while others wait in idle. If you implement this, your workers run safely without caring about how the application is deployed.

\* Stop the background workers (set `AbpBackgroundWorkerOptions.IsEnabled` to `false`) in all application instances except one of them, so only the single instance runs the workers.

\* Stop the background workers (set `AbpBackgroundWorkerOptions.IsEnabled` to `false`) in all application instances and create a dedicated application (maybe a console application running in its own container or a Windows Service running in the background) to execute all the background tasks. This can be a good option if your background workers consume high system resources (CPU, RAM or Disk), so you can deploy that background application to a dedicated server and your background tasks don't affect your application's performance.

## Integrations

Background worker system is extensible and you can change the default background worker manager with your own implementation or on of the pre-built integrations.

See pre-built worker manager alternatives:

\* [Quartz Background Worker Manager](Background-Workers-Quartz.md)

\* [Hangfire Background Worker Manager](Background-Workers-Hangfire.md)

## See Also

\* [Background Jobs](Background-Jobs.md)

### Quartz Integration

# Quartz Background Worker Manager

[Quartz](https://www.quartz-scheduler.net/) is an advanced background worker manager. You can integrate Quartz with the ABP Framework to use it instead of the [default background worker manager](Background-Workers.md). ABP simply integrates quartz.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.BackgroundWorkers.Quartz

````

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.BackgroundWorkers.Quartz](https://www.nuget.org/packages/Volo.Abp.BackgroundWorkers.Quartz) NuGet package to your project:

````

Install-Package Volo.Abp.BackgroundWorkers.Quartz

````

2. Add the `AbpBackgroundWorkersQuartzModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundWorkersQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

> Quartz background worker integration provided `QuartzPeriodicBackgroundWorkerAdapter` to adapt `PeriodicBackgroundWorkerBase` and `AsyncPeriodicBackgroundWorkerBase` derived class. So, you can still fllow the [background workers document](Background-Workers.md) to define the background worker.

## Configuration

See [Configuration](Background-Jobs-Quartz#Configuration).

## Create a Background Worker

A background work is a class that derives from the `QuartzBackgroundWorkerBase` base class. for example. A simple worker class is shown below:

```` csharp

public class MyLogWorker : QuartzBackgroundWorkerBase

{

public MyLogWorker()

{

JobDetail = JobBuilder.Create<MyLogWorker>().WithIdentity(nameof(MyLogWorker)).Build();

Trigger = TriggerBuilder.Create().WithIdentity(nameof(MyLogWorker)).StartNow().Build();

}

public override Task Execute(IJobExecutionContext context)

{

Logger.LogInformation("Executed MyLogWorker..!");

return Task.CompletedTask;

}

}

````

We simply implemented the Execute method to write a log. The background worker is a **\*\*singleton by default\*\***. If you want, you can also implement a [dependency interface](Dependency-Injection#DependencyInterfaces) to register it as another life cycle.

> Tips: Add identity to background workers is a best practice,because quartz distinguishes different jobs based on identity.

## Add to BackgroundWorkerManager

Default background workers are **\*\*automatically\*\*** added to the BackgroundWorkerManager when the application is **\*\*initialized\*\***. You can set `AutoRegister` property value to `false`,if you want to add it manually:

```` csharp

public class MyLogWorker : QuartzBackgroundWorkerBase

{

public MyLogWorker()

{

AutoRegister = false;

JobDetail = JobBuilder.Create<MyLogWorker>().WithIdentity(nameof(MyLogWorker)).Build();

Trigger = TriggerBuilder.Create().WithIdentity(nameof(MyLogWorker)).StartNow().Build();

}

public override Task Execute(IJobExecutionContext context)

{

Logger.LogInformation("Executed MyLogWorker..!");

return Task.CompletedTask;

}

}

````

If you want to globally disable auto add worker, you can global disable via `AbpBackgroundWorkerQuartzOptions` options:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundWorkersQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundWorkerQuartzOptions>(options =>

{

options.IsAutoRegisterEnabled = false;

});

}

}

```

## Advanced topics

### Customize ScheduleJob

Assume you have a worker executes every 10 minutes,but because server is unavailable for 30 minutes, 3 executions are missed. You want to execute all missed times after the server is available. You should define your background worker like this:

```csharp

public class MyLogWorker : QuartzBackgroundWorkerBase

{

public MyLogWorker()

{

JobDetail = JobBuilder.Create<MyLogWorker>().WithIdentity(nameof(MyLogWorker)).Build();

Trigger = TriggerBuilder.Create().WithIdentity(nameof(MyLogWorker)).WithSimpleSchedule(s=>s.WithIntervalInMinutes(1).RepeatForever().WithMisfireHandlingInstructionIgnoreMisfires()).Build();

ScheduleJob = async scheduler =>

{

if (!await scheduler.CheckExists(JobDetail.Key))

{

await scheduler.ScheduleJob(JobDetail, Trigger);

}

};

}

public override Task Execute(IJobExecutionContext context)

{

Logger.LogInformation("Executed MyLogWorker..!");

return Task.CompletedTask;

}

}

```

In the example we defined the worker execution interval to be 10 minutes and set `WithMisfireHandlingInstructionIgnoreMisfires`. we customized `ScheduleJob` and add worker to quartz only when the background worker does not exist.

### More

Please see Quartz's [documentation](https://www.quartz-scheduler.net/documentation/index.html) for more information.

### Hangfire Integration

# Hangfire Background Worker Manager

[Hangfire](https://www.hangfire.io/) is an advanced background jobs and worker manager. You can integrate Hangfire with the ABP Framework to use it instead of the [default background worker manager](Background-Workers.md).

The major advantage is that you can use the same server farm to manage your Background Jobs and Workers, as well as leverage the advanced scheduling that is available from Hangfire for [Recurring Jobs](https://docs.hangfire.io/en/latest/background-methods/performing-recurrent-tasks.html?highlight=recurring), aka Background Workers.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.BackgroundWorkers.Hangfire

````

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.BackgroundWorkers.Hangfire](https://www.nuget.org/packages/Volo.Abp.BackgroundWorkers.Hangfire) NuGet package to your project:

````

Install-Package Volo.Abp.BackgroundWorkers.Hangfire

````

2. Add the `AbpBackgroundWorkersHangfireModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundWorkersHangfireModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

> Hangfire background worker integration provides an adapter `HangfirePeriodicBackgroundWorkerAdapter` to automatically load any `PeriodicBackgroundWorkerBase` and `AsyncPeriodicBackgroundWorkerBase` derived classes as `IHangfireBackgroundWorker` instances. This allows you to still to easily switch over to use Hangfire as the background manager even you have existing background workers that are based on the [default background workers implementation](Background-Workers.md).

## Configuration

You can install any storage for Hangfire. The most common one is SQL Server (see the [Hangfire.SqlServer](https://www.nuget.org/packages/Hangfire.SqlServer) NuGet package).

After you have installed these NuGet packages, you need to configure your project to use Hangfire.

1.First, we change the `Module` class (example: `<YourProjectName>HttpApiHostModule`) to add Hangfire configuration of the storage and connection string in the `ConfigureServices` method:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

var hostingEnvironment = context.Services.GetHostingEnvironment();

//... other configarations.

ConfigureHangfire(context, configuration);

}

private void ConfigureHangfire(ServiceConfigurationContext context, IConfiguration configuration)

{

context.Services.AddHangfire(config =>

{

config.UseSqlServerStorage(configuration.GetConnectionString("Default"));

});

}

````

> You have to configure a storage for Hangfire.

2. If you want to use hangfire's dashboard, you can add `UseAbpHangfireDashboard` call in the `OnApplicationInitialization` method in `Module` class

````csharp

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

// ... others

app.UseAbpHangfireDashboard(); //should add to the request pipeline before the app.UseConfiguredEndpoints()

app.UseConfiguredEndpoints();

}

````

## Create a Background Worker

`HangfireBackgroundWorkerBase` is an easy way to create a background worker.

```` csharp

public class MyLogWorker : HangfireBackgroundWorkerBase

{

public MyLogWorker()

{

RecurringJobId = nameof(MyLogWorker);

CronExpression = Cron.Daily();

}

public override Task DoWorkAsync(CancellationToken cancellationToken = default)

{

Logger.LogInformation("Executed MyLogWorker..!");

return Task.CompletedTask;

}

}

````

\* **\*\*RecurringJobId\*\*** Is an optional parameter, see [Hangfire document](https://docs.hangfire.io/en/latest/background-methods/performing-recurrent-tasks.html)

\* **\*\*CronExpression\*\*** Is a CRON expression, see [CRON expression](https://en.wikipedia.org/wiki/Cron#CRON\_expression)

> You can directly implement the `IHangfireBackgroundWorker`, but `HangfireBackgroundWorkerBase` provides some useful properties like Logger.

### UnitOfWork

```csharp

public class MyLogWorker : HangfireBackgroundWorkerBase, IMyLogWorker

{

public MyLogWorker()

{

RecurringJobId = nameof(MyLogWorker);

CronExpression = Cron.Daily();

}

public override Task DoWorkAsync(CancellationToken cancellationToken = default)

{

using (var uow = LazyServiceProvider.LazyGetRequiredService<IUnitOfWorkManager>().Begin())

{

Logger.LogInformation("Executed MyLogWorker..!");

return Task.CompletedTask;

}

}

}

```

## Register BackgroundWorkerManager

After creating a background worker class, you should add it to the `IBackgroundWorkerManager`. The most common place is the `OnApplicationInitializationAsync` method of your module class:

```` csharp

[DependsOn(typeof(AbpBackgroundWorkersModule))]

public class MyModule : AbpModule

{

public override async Task OnApplicationInitializationAsync(

ApplicationInitializationContext context)

{

await context.AddBackgroundWorkerAsync<MyLogWorker>();

}

}

````

`context.AddBackgroundWorkerAsync(...)` is a shortcut extension method for the expression below:

```` csharp

context.ServiceProvider

.GetRequiredService<IBackgroundWorkerManager>()

.AddAsync(

context

.ServiceProvider

.GetRequiredService<MyLogWorker>()

);

````

So, it resolves the given background worker and adds to the `IBackgroundWorkerManager`.

While we generally add workers in `OnApplicationInitializationAsync`, there are no restrictions on that. You can inject `IBackgroundWorkerManager` anywhere and add workers at runtime. Background worker manager will stop and release all the registered workers when your application is being shut down.

## BLOB Storing

### BLOB Storing System

# BLOB Storing

It is typical to **\*\*store file contents\*\*** in an application and read these file contents on need. Not only files, but you may also need to save various types of **\*\*large binary objects\*\***, a.k.a. [BLOB](https://en.wikipedia.org/wiki/Binary\_large\_object)s, into a **\*\*storage\*\***. For example, you may want to save user profile pictures.

A BLOB is a typically **\*\*byte array\*\***. There are various places to store a BLOB item; storing in the local file system, in a shared database or on the [Azure BLOB storage](https://azure.microsoft.com/en-us/services/storage/blobs/) can be options.

The ABP Framework provides an abstraction to work with BLOBs and provides some pre-built storage providers that you can easily integrate to. Having such an abstraction has some benefits;

\* You can **\*\*easily integrate\*\*** to your favorite BLOB storage provides with a few lines of configuration.

\* You can then **\*\*easily change\*\*** your BLOB storage without changing your application code.

\* If you want to create **\*\*reusable application modules\*\***, you don't need to make assumption about how the BLOBs are stored.

ABP BLOB Storage system is also compatible to other ABP Framework features like [multi-tenancy](Multi-Tenancy.md).

## BLOB Storage Providers

The ABP Framework has already the following storage provider implementations:

\* [File System](Blob-Storing-File-System.md): Stores BLOBs in a folder of the local file system, as standard files.

\* [Database](Blob-Storing-Database.md): Stores BLOBs in a database.

\* [Azure](Blob-Storing-Azure.md): Stores BLOBs on the [Azure BLOB storage](https://azure.microsoft.com/en-us/services/storage/blobs/).

\* [Aliyun](Blob-Storing-Aliyun.md): Stores BLOBs on the [Aliyun Storage Service](https://help.aliyun.com/product/31815.html).

\* [Minio](Blob-Storing-Minio.md): Stores BLOBs on the [MinIO Object storage](https://min.io/).

\* [Aws](Blob-Storing-Aws.md): Stores BLOBs on the [Amazon Simple Storage Service](https://aws.amazon.com/s3/).

More providers will be implemented by the time. You can [request](https://github.com/abpframework/abp/issues/new) it for your favorite provider or [create it yourself](Blob-Storing-Custom-Provider.md) and [contribute](Contribution/Index.md) to the ABP Framework.

Multiple providers **\*\*can be used together\*\*** by the help of the **\*\*container system\*\***, where each container can uses a different provider.

> BLOB storing system can not work unless you **\*\*configure a storage provider\*\***. Refer to the linked documents for the storage provider configurations.

## Installation

[Volo.Abp.BlobStoring](https://www.nuget.org/packages/Volo.Abp.BlobStoring) is the main package that defines the BLOB storing services. You can use this package to use the BLOB Storing system without depending a specific storage provider.

Use the ABP CLI to add this package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), if you haven't installed it.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BlobStoring` package.

\* Run `abp add-package Volo.Abp.BlobStoring` command.

If you want to do it manually, install the [Volo.Abp.BlobStoring](https://www.nuget.org/packages/Volo.Abp.BlobStoring) NuGet package to your project and add `[DependsOn(typeof(AbpBlobStoringModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## The IBlobContainer

`IBlobContainer` is the main interface to store and read BLOBs. Your application may have multiple containers and each container can be separately configured. But, there is a **\*\*default container\*\*** that can be simply used by [injecting](Dependency-Injection.md) the `IBlobContainer`.

**\*\*Example: Simply save and read bytes of a named BLOB\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.BlobStoring;

using Volo.Abp.DependencyInjection;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IBlobContainer \_blobContainer;

public MyService(IBlobContainer blobContainer)

{

\_blobContainer = blobContainer;

}

public async Task SaveBytesAsync(byte[] bytes)

{

await \_blobContainer.SaveAsync("my-blob-1", bytes);

}

public async Task<byte[]> GetBytesAsync()

{

return await \_blobContainer.GetAllBytesOrNullAsync("my-blob-1");

}

}

}

````

This service saves the given bytes with the `my-blob-1` name and then gets the previously saved bytes with the same name.

> A BLOB is a named object and **\*\*each BLOB should have a unique name\*\***, which is an arbitrary string.

`IBlobContainer` can work with `Stream` and `byte[]` objects, which will be detailed in the next sections.

### Saving BLOBs

`SaveAsync` method is used to save a new BLOB or replace an existing BLOB. It can save a `Stream` by default, but there is a shortcut extension method to save byte arrays.

`SaveAsync` gets the following parameters:

\* **\*\*name\*\*** (string): Unique name of the BLOB.

\* **\*\*stream\*\*** (Stream) or **\*\*bytes\*\*** (byte[]): The stream to read the BLOB content or a byte array.

\* **\*\*overrideExisting\*\*** (bool): Set `true` to replace the BLOB content if it does already exists. Default value is `false` and throws `BlobAlreadyExistsException` if there is already a BLOB in the container with the same name.

### Reading/Getting BLOBs

\* `GetAsync`: Only gets a BLOB name and returns a `Stream` object that can be used to read the BLOB content. Always **\*\*dispose the stream\*\*** after using it. This method throws exception, if it can not find the BLOB with the given name.

\* `GetOrNullAsync`: In opposite to the `GetAsync` method, this one returns `null` if there is no BLOB found with the given name.

\* `GetAllBytesAsync`: Returns a `byte[]` instead of a `Stream`. Still throws exception if can not find the BLOB with the given name.

\* `GetAllBytesOrNullAsync`: In opposite to the `GetAllBytesAsync` method, this one returns `null` if there is no BLOB found with the given name.

### Deleting BLOBs

`DeleteAsync` method gets a BLOB name and deletes the BLOB data. It doesn't throw any exception if given BLOB was not found. Instead, it returns a `bool` indicating that the BLOB was actually deleted or not, if you care about it.

### Other Methods

\* `ExistsAsync` method simply checks if there is a BLOB in the container with the given name.

### About Naming the BLOBs

There is not a rule for naming the BLOBs. A BLOB name is just a string that is unique per container (and per tenant - see the "*\*Multi-Tenancy\**" section). However, different storage providers may conventionally implement some practices. For example, the [File System Provider](Blob-Storing-File-System.md) use directory separators (`/`) and file extensions in your BLOB name (if your BLOB name is `images/common/x.png` then it is saved as `x.png` in the `images/common` folder inside the root container folder).

## Typed IBlobContainer

Typed BLOB container system is a way of creating and managing **\*\*multiple containers\*\*** in an application;

\* **\*\*Each container is separately stored\*\***. That means the BLOB names should be unique in a container and two BLOBs with the same name can live in different containers without effecting each other.

\* **\*\*Each container can be separately configured\*\***, so each container can use a different storage provider based on your configuration.

To create a typed container, you need to create a simple class decorated with the `BlobContainerName` attribute:

````csharp

using Volo.Abp.BlobStoring;

namespace AbpDemo

{

[BlobContainerName("profile-pictures")]

public class ProfilePictureContainer

{

}

}

````

> If you don't use the `BlobContainerName` attribute, ABP Framework uses the full name of the class (with namespace), but it is always recommended to use a container name which is stable and does not change even if you rename the class.

Once you create the container class, you can inject `IBlobContainer<T>` for your container type.

**\*\*Example: An [**application service**](Application-Services.md) to save and read profile picture of the [**current user**](CurrentUser.md)\*\***

````csharp

[Authorize]

public class ProfileAppService : ApplicationService

{

private readonly IBlobContainer<ProfilePictureContainer> \_blobContainer;

public ProfileAppService(IBlobContainer<ProfilePictureContainer> blobContainer)

{

\_blobContainer = blobContainer;

}

public async Task SaveProfilePictureAsync(byte[] bytes)

{

var blobName = CurrentUser.GetId().ToString();

await \_blobContainer.SaveAsync(blobName, bytes);

}

public async Task<byte[]> GetProfilePictureAsync()

{

var blobName = CurrentUser.GetId().ToString();

return await \_blobContainer.GetAllBytesOrNullAsync(blobName);

}

}

````

`IBlobContainer<T>` has the same methods with the `IBlobContainer`.

> It is a good practice to **\*\*always use a typed container while developing re-usable modules\*\***, so the final application can configure the provider for your container without effecting the other containers.

### The Default Container

If you don't use the generic argument and directly inject the `IBlobContainer` (as explained before), you get the default container. Another way of injecting the default container is using `IBlobContainer<DefaultContainer>`, which returns exactly the same container.

The name of the default container is `default`.

### Named Containers

Typed containers are just shortcuts for named containers. You can inject and use the `IBlobContainerFactory` to get a BLOB container by its name:

````csharp

public class ProfileAppService : ApplicationService

{

private readonly IBlobContainer \_blobContainer;

public ProfileAppService(IBlobContainerFactory blobContainerFactory)

{

\_blobContainer = blobContainerFactory.Create("profile-pictures");

}

//...

}

````

## IBlobContainerFactory

`IBlobContainerFactory` is the service that is used to create the BLOB containers. One example was shown above.

**\*\*Example: Create a container by name\*\***

````csharp

var blobContainer = blobContainerFactory.Create("profile-pictures");

````

**\*\*Example: Create a container by type\*\***

````csharp

var blobContainer = blobContainerFactory.Create<ProfilePictureContainer>();

````

> You generally don't need to use the `IBlobContainerFactory` since it is used internally, when you inject a `IBlobContainer` or `IBlobContainer<T>`.

## Configuring the Containers

Containers should be configured before using them. The most fundamental configuration is to **\*\*select a BLOB storage provider\*\*** (see the "*\*BLOB Storage Providers\**" section above).

`AbpBlobStoringOptions` is the [options class](Options.md) to configure the containers. You can configure the options inside the `ConfigureServices` method of your [module](Module-Development-Basics.md).

### Configure a Single Container

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.Configure<ProfilePictureContainer>(container =>

{

//TODO...

});

});

````

This example configures the `ProfilePictureContainer`. You can also configure by the container name:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.Configure("profile-pictures", container =>

{

//TODO...

});

});

````

### Configure the Default Container

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

//TODO...

});

});

````

> There is a special case about the default container; If you don't specify a configuration for a container, it **\*\*fallbacks to the default container configuration\*\***. This is a good way to configure defaults for all containers and specialize configuration for a specific container when needed.

### Configure All Containers

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureAll((containerName, containerConfiguration) =>

{

//TODO...

});

});

````

This is a way to configure all the containers.

> The main difference from configuring the default container is that `ConfigureAll` overrides the configuration even if it was specialized for a specific container.

## Multi-Tenancy

If your application is set as multi-tenant, the BLOB Storage system **\*\*works seamlessly with the [**multi-tenancy**](Multi-Tenancy.md)\*\***. All the providers implement multi-tenancy as a standard feature. They **\*\*isolate BLOBs\*\*** of different tenants from each other, so they can only access to their own BLOBs. It means you can use the **\*\*same BLOB name for different tenants\*\***.

If your application is multi-tenant, you may want to control **\*\*multi-tenancy behavior\*\*** of the containers individually. For example, you may want to **\*\*disable multi-tenancy\*\*** for a specific container, so the BLOBs inside it will be **\*\*available to all the tenants\*\***. This is a way to share BLOBs among all tenants.

**\*\*Example: Disable multi-tenancy for a specific container\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.Configure<ProfilePictureContainer>(container =>

{

container.IsMultiTenant = false;

});

});

````

> If your application is not multi-tenant, no worry, it works as expected. You don't need to configure the `IsMultiTenant` option.

## Extending the BLOB Storing System

Most of the times, you won't need to customize the BLOB storage system except [creating a custom BLOB storage provider](Blob-Storing-Custom-Provider.md). However, you can replace any service (injected via [dependency injection](Dependency-Injection.md)), if you need. Here, some other services not mentioned above, but you may want to know:

\* `IBlobProviderSelector` is used to get a `IBlobProvider` instance by a container name. Default implementation (`DefaultBlobProviderSelector`) selects the provider using the configuration.

\* `IBlobContainerConfigurationProvider` is used to get the `BlobContainerConfiguration` for a given container name. Default implementation (`DefaultBlobContainerConfigurationProvider`) gets the configuration from the `AbpBlobStoringOptions` explained above.

## BLOB Storing vs File Management System

Notice that BLOB storing is not a file management system. It is a low level system that is used to save, get and delete named BLOBs. It doesn't provide a hierarchical structure like directories, you may expect from a typical file system.

If you want to create folders and move files between folders, assign permissions to files and share files between users then you need to implement your own application on top of the BLOB Storage system.

## See Also

\* [Creating a custom BLOB storage provider](Blob-Storing-Custom-Provider.md)

### Storage Providers

#### File System Provider

# BLOB Storing File System Provider

File System Storage Provider is used to store BLOBs in the local file system as standard files inside a folder.

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to configure containers to use the file system.

## Installation

Use the ABP CLI to add [Volo.Abp.BlobStoring.FileSystem](https://www.nuget.org/packages/Volo.Abp.BlobStoring.FileSystem) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BlobStoring.FileSystem` package.

\* Run `abp add-package Volo.Abp.BlobStoring.FileSystem` command.

If you want to do it manually, install the [Volo.Abp.BlobStoring.FileSystem](https://www.nuget.org/packages/Volo.Abp.BlobStoring.FileSystem) NuGet package to your project and add `[DependsOn(typeof(AbpBlobStoringFileSystemModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

Configuration is done in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class, as explained in the [BLOB Storing document](Blob-Storing.md).

**\*\*Example: Configure to use the File System storage provider by default\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseFileSystem(fileSystem =>

{

fileSystem.BasePath = "C:\\my-files";

});

});

});

````

`UseFileSystem` extension method is used to set the File System Provider for a container and configure the file system options.

> See the [BLOB Storing document](Blob-Storing.md) to learn how to configure this provider for a specific container.

### Options

\* **\*\*BasePath\*\*** (string): The base folder path to store BLOBs. It is required to set this option.

\* **\*\*AppendContainerNameToBasePath\*\*** (bool; default: `true`): Indicates whether to create a folder with the container name inside the base folder. If you store multiple containers in the same `BaseFolder`, leave this as `true`. Otherwise, you can set it to `false` if you don't like an unnecessarily deeper folder hierarchy.

## File Path Calculation

File System Provider organizes BLOB files inside folders and implements some conventions. The full path of a BLOB file is determined by the following rules by default:

\* It starts with the `BasePath` configured as shown above.

\* Appends `host` folder if [current tenant](Multi-Tenancy.md) is `null` (or multi-tenancy is disabled for the container - see the [BLOB Storing document](Blob-Storing.md) to learn how to disable multi-tenancy for a container).

\* Appends `tenants/<tenant-id>` folder if current tenant is not `null`.

\* Appends the container's name if `AppendContainerNameToBasePath` is `true`. If container name contains `/`, this will result with nested folders.

\* Appends the BLOB name. If the BLOB name contains `/` it creates folders. If the BLOB name contains `.` it will have a file extension.

## Extending the File System BLOB Provider

\* `FileSystemBlobProvider` is the main service that implements the File System storage. You can inherit from this class and [override](Customizing-Application-Modules-Overriding-Services.md) methods to customize it.

\* The `IBlobFilePathCalculator` service is used to calculate the file paths. Default implementation is the `DefaultBlobFilePathCalculator`. You can replace/override it if you want to customize the file path calculation.

#### Database Provider

# BLOB Storing Database Provider

BLOB Storing Database Storage Provider can store BLOBs in a relational or non-relational database.

There are two database providers implemented;

\* [Volo.Abp.BlobStoring.Database.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.EntityFrameworkCore) package implements for [EF Core](Entity-Framework-Core.md), so it can store BLOBs in [any DBMS supported](https://docs.microsoft.com/en-us/ef/core/providers/) by the EF Core.

\* [Volo.Abp.BlobStoring.Database.MongoDB](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.MongoDB) package implements for [MongoDB](MongoDB.md).

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to configure containers to use a database as the storage provider.

## Installation

### Automatic Installation

If you've created your solution based on the [application startup template](Startup-Templates/Application.md), you can use the `abp add-module` [CLI](CLI.md) command to automatically add related packages to your solution.

Open a command prompt (terminal) in the folder containing your solution (`.sln`) file and run the following command:

````bash

abp add-module Volo.Abp.BlobStoring.Database

````

This command adds all the NuGet packages to corresponding layers of your solution. If you are using EF Core, it adds necessary configuration, adds a new database migration and updates the database.

### Manual Installation

Here, all the NuGet packages defined by this provider;

\* [Volo.Abp.BlobStoring.Database.Domain.Shared](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.Domain.Shared)

\* [Volo.Abp.BlobStoring.Database.Domain](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.Domain)

\* [Volo.Abp.BlobStoring.Database.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.EntityFrameworkCore)

\* [Volo.Abp.BlobStoring.Database.MongoDB](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.MongoDB)

You can only install Volo.Abp.BlobStoring.Database.EntityFrameworkCore or Volo.Abp.BlobStoring.Database.MongoDB (based on your preference) since they depends on the other packages.

After installation, add `DepenedsOn` attribute to your related [module](Module-Development-Basics.md). Here, the list of module classes defined by the related NuGet packages listed above:

\* `BlobStoringDatabaseDomainModule`

\* `BlobStoringDatabaseDomainSharedModule`

\* `BlobStoringDatabaseEntityFrameworkCoreModule`

\* `BlobStoringDatabaseMongoDbModule`

Whenever you add a NuGet package to a project, also add the module class dependency.

If you are using EF Core, you also need to configure your **\*\*Migration DbContext\*\*** to add BLOB storage tables to your database schema. Call `builder.ConfigureBlobStoring()` extension method inside the `OnModelCreating` method to include mappings to your DbContext. Then you can use the standard `Add-Migration` and `Update-Database` [commands](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/) to create necessary tables in your database.

## Configuration

### Connection String

If you will use your `Default` connection string, you don't need to any additional configuration.

If you want to use a separate database for BLOB storage, use the `AbpBlobStoring` as the [connection string](Connection-Strings.md) name in your configuration file (`appsettings.json`). In this case, also read the [EF Core Migrations](Entity-Framework-Core-Migrations.md) document to learn how to create and use a different database for a desired module.

### Configuring the Containers

If you are using only the database storage provider, you don't need to manually configure it, since it is automatically done. If you are using multiple storage providers, you may want to configure it.

Configuration is done in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class, as explained in the [BLOB Storing document](Blob-Storing.md).

**\*\*Example: Configure to use the database storage provider by default\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseDatabase();

});

});

````

> See the [BLOB Storing document](Blob-Storing.md) to learn how to configure this provider for a specific container.

## Additional Information

It is expected to use the [BLOB Storing services](Blob-Storing.md) to use the BLOB storing system. However, if you want to work on the database tables/entities, you can use the following information.

### Entities

Entities defined for this module:

\* `DatabaseBlobContainer` (aggregate root) represents a container stored in the database.

\* `DatabaseBlob` (aggregate root) represents a BLOB in the database.

See the [entities document](Entities.md) to learn what is an entity and aggregate root.

### Repositories

\* `IDatabaseBlobContainerRepository`

\* `IDatabaseBlobRepository`

You can also use `IRepository<DatabaseBlobContainer, Guid>` and `IRepository<DatabaseBlob, Guid>` to take the power of IQueryable. See the [repository document](Repositories.md) for more.

### Other Services

\* `DatabaseBlobProvider` is the main service that implements the database BLOB storage provider, if you want to override/replace it via [dependency injection](Dependency-Injection.md) (don't replace `IBlobProvider` interface, but replace `DatabaseBlobProvider` class).

#### Azure Provider

# BLOB Storing Azure Provider

BLOB Storing Azure Provider can store BLOBs in [Azure Blob storage](https://azure.microsoft.com/en-us/services/storage/blobs/).

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to configure containers to use a Azure BLOB as the storage provider.

## Installation

Use the ABP CLI to add [Volo.Abp.BlobStoring.Azure](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Azure) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BlobStoring.Azure` package.

\* Run `abp add-package Volo.Abp.BlobStoring.Azure` command.

If you want to do it manually, install the [Volo.Abp.BlobStoring.Azure](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Azure) NuGet package to your project and add `[DependsOn(typeof(AbpBlobStoringAzureModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

Configuration is done in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class, as explained in the [BLOB Storing document](Blob-Storing.md).

**\*\*Example: Configure to use the azure storage provider by default\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseAzure(azure =>

{

azure.ConnectionString = "your azure connection string";

azure.ContainerName = "your azure container name";

azure.CreateContainerIfNotExists = true;

});

});

});

````

> See the [BLOB Storing document](Blob-Storing.md) to learn how to configure this provider for a specific container.

### Options

\* **\*\*ConnectionString\*\*** (string): A connection string includes the authorization information required for your application to access data in an Azure Storage account at runtime using Shared Key authorization. Please refer to Azure documentation: https://docs.microsoft.com/en-us/azure/storage/common/storage-configure-connection-string

\* **\*\*ContainerName\*\*** (string): You can specify the container name in azure. If this is not specified, it uses the name of the BLOB container defined with the `BlobContainerName` attribute (see the [BLOB storing document](Blob-Storing.md)). Please note that Azure has some **\*\*rules for naming containers\*\***. A container name must be a valid DNS name, conforming to the [following naming rules](https://docs.microsoft.com/en-us/rest/api/storageservices/naming-and-referencing-containers--blobs--and-metadata#container-names):

\* Container names must start or end with a letter or number, and can contain only letters, numbers, and the dash (-) character.

\* Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.

\* All letters in a container name must be **\*\*lowercase\*\***.

\* Container names must be from **\*\*3\*\*** through **\*\*63\*\*** characters long.

\* **\*\*CreateContainerIfNotExists\*\*** (bool): Default value is `false`, If a container does not exist in azure, `AzureBlobProvider` will try to create it.

## Azure Blob Name Calculator

Azure Blob Provider organizes BLOB name and implements some conventions. The full name of a BLOB is determined by the following rules by default:

\* Appends `host` string if [current tenant](Multi-Tenancy.md) is `null` (or multi-tenancy is disabled for the container - see the [BLOB Storing document](Blob-Storing.md) to learn how to disable multi-tenancy for a container).

\* Appends `tenants/<tenant-id>` string if current tenant is not `null`.

\* Appends the BLOB name.

## Other Services

\* `AzureBlobProvider` is the main service that implements the Azure BLOB storage provider, if you want to override/replace it via [dependency injection](Dependency-Injection.md) (don't replace `IBlobProvider` interface, but replace `AzureBlobProvider` class).

\* `IAzureBlobNameCalculator` is used to calculate the full BLOB name (that is explained above). It is implemented by the `DefaultAzureBlobNameCalculator` by default.

#### Aliyun Provider

# BLOB Storing Aliyun Provider

BLOB Storing Aliyun Provider can store BLOBs in [Aliyun Blob storage](https://help.aliyun.com/product/31815.html).

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to configure containers to use a Aliyun BLOB as the storage provider.

## Installation

Use the ABP CLI to add [Volo.Abp.BlobStoring.Aliyun](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Aliyun) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BlobStoring.Aliyun` package.

\* Run `abp add-package Volo.Abp.BlobStoring.Aliyun` command.

If you want to do it manually, install the [Volo.Abp.BlobStoring.Aliyun](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Aliyun) NuGet package to your project and add `[DependsOn(typeof(AbpBlobStoringAliyunModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

Configuration is done in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class, as explained in the [BLOB Storing document](Blob-Storing.md).

**\*\*Example: Configure to use the Aliyun storage provider by default\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseAliyun(aliyun =>

{

aliyun.AccessKeyId = "your aliyun access key id";

aliyun.AccessKeySecret = "your aliyun access key secret";

aliyun.Endpoint = "your oss endpoint";

aliyun.RegionId = "your sts region id";

aliyun.RoleArn = "the arn of ram role";

aliyun.RoleSessionName = "the name of the certificate";

aliyun.Policy = "policy";

aliyun.DurationSeconds = "expiration date";

aliyun.ContainerName = "your aliyun container name";

aliyun.CreateContainerIfNotExists = true;

});

});

});

````

> See the [BLOB Storing document](Blob-Storing.md) to learn how to configure this provider for a specific container.

### Options

\* **\*\*AccessKeyId\*\*** ([NotNull]string): AccessKey is the key to access the Alibaba Cloud API. It has full permissions for the account. Please keep it safe! Recommend to follow [Alibaba Cloud security best practicess](https://help.aliyun.com/document\_detail/102600.html),Use RAM sub-user AccessKey to call API.

\* **\*\*AccessKeySecret\*\*** ([NotNull]string): Same as above.

\* **\*\*Endpoint\*\*** ([NotNull]string): Endpoint is the external domain name of OSS. See the [document](https://help.aliyun.com/document\_detail/31837.html) for details.

\* **\*\*UseSecurityTokenService\*\*** (bool): Use [STS temporary credentials](https://help.aliyun.com/document\_detail/100624.html) to access OSS services,default: `false`.

\* **\*\*RegionId\*\*** (string): Access address of STS service. See the [document](https://help.aliyun.com/document\_detail/66053.html) for details.

\* **\*\*RoleArn\*\*** ([NotNull]string): STS required role ARN. See the [document](https://help.aliyun.com/document\_detail/100624.html) for details.

\* **\*\*RoleSessionName\*\*** ([NotNull]string): Used to identify the temporary access credentials, it is recommended to use different application users to distinguish.

\* **\*\*Policy\*\*** (string): Additional permission restrictions. See the [document](https://help.aliyun.com/document\_detail/100680.html) for details.

\* **\*\*DurationSeconds\*\*** (int): Validity period(s) of a temporary access certificate,minimum is 900 and the maximum is 3600.

\* **\*\*ContainerName\*\*** (string): You can specify the container name in Aliyun. If this is not specified, it uses the name of the BLOB container defined with the `BlobContainerName` attribute (see the [BLOB storing document](Blob-Storing.md)). Please note that Aliyun has some **\*\*rules for naming containers\*\***. A container name must be a valid DNS name, conforming to the [following naming rules](https://help.aliyun.com/knowledge\_detail/39668.html):

\* Container names must start or end with a letter or number, and can contain only letters, numbers, and the dash (-) character.

\* Container names Must start and end with lowercase letters and numbers.

\* Container names must be from **\*\*3\*\*** through **\*\*63\*\*** characters long.

\* **\*\*CreateContainerIfNotExists\*\*** (bool): Default value is `false`, If a container does not exist in Aliyun, `AliyunBlobProvider` will try to create it.

\* **\*\*TemporaryCredentialsCacheKey\*\*** (bool): The cache key of STS credentials.

## Aliyun Blob Name Calculator

Aliyun Blob Provider organizes BLOB name and implements some conventions. The full name of a BLOB is determined by the following rules by default:

\* Appends `host` string if [current tenant](Multi-Tenancy.md) is `null` (or multi-tenancy is disabled for the container - see the [BLOB Storing document](Blob-Storing.md) to learn how to disable multi-tenancy for a container).

\* Appends `tenants/<tenant-id>` string if current tenant is not `null`.

\* Appends the BLOB name.

## Other Services

\* `AliyunBlobProvider` is the main service that implements the Aliyun BLOB storage provider, if you want to override/replace it via [dependency injection](Dependency-Injection.md) (don't replace `IBlobProvider` interface, but replace `AliyunBlobProvider` class).

\* `IAliyunBlobNameCalculator` is used to calculate the full BLOB name (that is explained above). It is implemented by the `DefaultAliyunBlobNameCalculator` by default.

\* `IOssClientFactory` is used create OSS client. It is implemented by the `DefaultOssClientFactory` by default. You can override/replace it,if you want customize.

#### Minio Provider

# BLOB Storing Minio Provider

BLOB Storing Minio Provider can store BLOBs in [MinIO Object storage](https://min.io/).

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to configure containers to use a Minio BLOB as the storage provider.

## Installation

Use the ABP CLI to add [Volo.Abp.BlobStoring.Minio](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Minio) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BlobStoring.Minio` package.

\* Run `abp add-package Volo.Abp.BlobStoring.Minio` command.

If you want to do it manually, install the [Volo.Abp.BlobStoring.Minio](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Minio) NuGet package to your project and add `[DependsOn(typeof(AbpBlobStoringMinioModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

Configuration is done in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class, as explained in the [BLOB Storing document](Blob-Storing.md).

**\*\*Example: Configure to use the minio storage provider by default\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseMinio(minio =>

{

minio.EndPoint = "your minio endPoint";

minio.AccessKey = "your minio accessKey";

minio.SecretKey = "your minio secretKey";

minio.BucketName = "your minio bucketName";

});

});

});

````

> See the [BLOB Storing document](Blob-Storing.md) to learn how to configure this provider for a specific container.

### Options

\* **\*\*EndPoint\*\*** (string): URL to object storage service. Please refer to MinIO Client SDK for .NET: https://docs.min.io/docs/dotnet-client-quickstart-guide.html

\* **\*\*AccessKey\*\*** (string): Access key is the user ID that uniquely identifies your account.

\* **\*\*SecretKey\*\*** (string): Secret key is the password to your account.

\* **\*\*BucketName\*\*** (string): You can specify the bucket name in MinIO. If this is not specified, it uses the name of the BLOB container defined with the `BlobContainerName` attribute (see the [BLOB storing document](Blob-Storing.md)).MinIO is the defacto standard for S3 compatibility, So MinIO has some **\*\*rules for naming bucket\*\***. The [following rules](https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html) apply for naming MinIO buckets:

\* Bucket names must be between **\*\*3\*\*** and **\*\*63\*\*** characters long.

\* Bucket names can consist only of **\*\*lowercase\*\*** letters, numbers, dots (.), and hyphens (-).

\* Bucket names must begin and end with a letter or number.

\* Bucket names must not be formatted as an IP address (for example, 192.168.5.4).

\* Bucket names can't begin with **\*\*xn--\*\*** (for buckets created after February 2020).

\* Bucket names must be unique within a partition.

\* Buckets used with Amazon S3 Transfer Acceleration can't have dots (.) in their names. For more information about transfer acceleration, see Amazon S3 Transfer Acceleration.

\* **\*\*WithSSL\*\*** (bool): Default value is `false`,Chain to MinIO Client object to use https instead of http.

\* **\*\*CreateContainerIfNotExists\*\*** (bool): Default value is `false`, If a bucket does not exist in minio, `MinioBlobProvider` will try to create it.

## Minio Blob Name Calculator

Minio Blob Provider organizes BLOB name and implements some conventions. The full name of a BLOB is determined by the following rules by default:

\* Appends `host` string if [current tenant](Multi-Tenancy.md) is `null` (or multi-tenancy is disabled for the container - see the [BLOB Storing document](Blob-Storing.md) to learn how to disable multi-tenancy for a container).

\* Appends `tenants/<tenant-id>` string if current tenant is not `null`.

\* Appends the BLOB name.

## Other Services

\* `MinioBlobProvider` is the main service that implements the Minio BLOB storage provider, if you want to override/replace it via [dependency injection](Dependency-Injection.md) (don't replace `IBlobProvider` interface, but replace `MinioBlobProvider` class).

\* `IMinioBlobNameCalculator` is used to calculate the full BLOB name (that is explained above). It is implemented by the `DefaultMinioBlobNameCalculator` by default.

#### AWS Provider

# BLOB Storing Aws Provider

BLOB Storing Aws Provider can store BLOBs in [Amazon Simple Storage Service](https://aws.amazon.com/s3/).

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to configure containers to use a Aws BLOB as the storage provider.

## Installation

Use the ABP CLI to add [Volo.Abp.BlobStoring.Aws](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Aws) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.BlobStoring.Aws` package.

\* Run `abp add-package Volo.Abp.BlobStoring.Aws` command.

If you want to do it manually, install the [Volo.Abp.BlobStoring.Aws](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Aws) NuGet package to your project and add `[DependsOn(typeof(AbpBlobStoringAwsModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

Configuration is done in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class, as explained in the [BLOB Storing document](Blob-Storing.md).

**\*\*Example: Configure to use the Aws storage provider by default\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseAws(Aws =>

{

Aws.AccessKeyId = "your Aws access key id";

Aws.SecretAccessKey = "your Aws access key secret";

Aws.UseCredentials = "set true to use credentials";

Aws.UseTemporaryCredentials = "set true to use temporary credentials";

Aws.UseTemporaryFederatedCredentials = "set true to use temporary federated credentials";

Aws.ProfileName = "the name of the profile to get credentials from";

Aws.ProfilesLocation = "the path to the aws credentials file to look at";

Aws.Region = "the system name of the service";

Aws.Name = "the name of the federated user";

Aws.Policy = "policy";

Aws.DurationSeconds = "expiration date";

Aws.ContainerName = "your Aws container name";

Aws.CreateContainerIfNotExists = true;

});

});

});

````

> See the [BLOB Storing document](Blob-Storing.md) to learn how to configure this provider for a specific container.

### Options

\* **\*\*AccessKeyId\*\*** (string): AWS Access Key ID.

\* **\*\*SecretAccessKey\*\*** (string): AWS Secret Access Key.

\* **\*\*UseCredentials\*\*** (bool): Use [credentials](https://docs.aws.amazon.com/AmazonS3/latest/dev/AuthUsingAcctOrUserCredentials.html) to access AWS services,default : `false`.

\* **\*\*UseTemporaryCredentials\*\*** (bool): Use [temporary credentials](https://docs.aws.amazon.com/AmazonS3/latest/dev/AuthUsingTempSessionToken.html) to access AWS services,default : `false`.

\* **\*\*UseTemporaryFederatedCredentials\*\*** (bool): Use [federated user temporary credentials](https://docs.aws.amazon.com/AmazonS3/latest/dev/AuthUsingTempFederationToken.html) to access AWS services, default : `false`.

\* **\*\*ProfileName\*\*** (string): The [name of the profile](https://docs.aws.amazon.com/sdk-for-net/v3/developer-guide/net-dg-config-creds.html) to get credentials from.

\* **\*\*ProfilesLocation\*\*** (string): The path to the aws credentials file to look at.

\* **\*\*Region\*\*** (string): The system name of the service.

\* **\*\*Policy\*\*** (string): An IAM policy in JSON format that you want to use as an inline session policy.

\* **\*\*DurationSeconds\*\*** (int): Validity period(s) of a temporary access certificate,minimum is 900 and the maximum is 3600. **\*\*note\*\***: Using sub-accounts operated OSS,if the value is 0.

\* **\*\*ContainerName\*\*** (string): You can specify the container name in Aws. If this is not specified, it uses the name of the BLOB container defined with the `BlobContainerName` attribute (see the [BLOB storing document](Blob-Storing.md)). Please note that Aws has some **\*\*rules for naming containers\*\***. A container name must be a valid DNS name, conforming to the [following naming rules](https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html):

\* Bucket names must be between **\*\*3\*\*** and **\*\*63\*\*** characters long.

\* Bucket names can consist only of **\*\*lowercase\*\*** letters, numbers, dots (.), and hyphens (-).

\* Bucket names must begin and end with a letter or number.

\* Bucket names must not be formatted as an IP address (for example, 192.168.5.4).

\* Bucket names can't begin with **\*\*xn--\*\*** (for buckets created after February 2020).

\* Bucket names must be unique within a partition.

\* Buckets used with Amazon S3 Transfer Acceleration can't have dots (.) in their names. For more information about transfer acceleration, see Amazon S3 Transfer Acceleration.

\* **\*\*CreateContainerIfNotExists\*\*** (bool): Default value is `false`, If a container does not exist in Aws, `AwsBlobProvider` will try to create it.

## Aws Blob Name Calculator

Aws Blob Provider organizes BLOB name and implements some conventions. The full name of a BLOB is determined by the following rules by default:

\* Appends `host` string if [current tenant](Multi-Tenancy.md) is `null` (or multi-tenancy is disabled for the container - see the [BLOB Storing document](Blob-Storing.md) to learn how to disable multi-tenancy for a container).

\* Appends `tenants/<tenant-id>` string if current tenant is not `null`.

\* Appends the BLOB name.

## Other Services

\* `AwsBlobProvider` is the main service that implements the Aws BLOB storage provider, if you want to override/replace it via [dependency injection](Dependency-Injection.md) (don't replace `IBlobProvider` interface, but replace `AwsBlobProvider` class).

\* `IAwsBlobNameCalculator` is used to calculate the full BLOB name (that is explained above). It is implemented by the `DefaultAwsBlobNameCalculator` by default.

\* `IAmazonS3ClientFactory` is used create OSS client. It is implemented by the `DefaultAmazonS3ClientFactory` by default. You can override/replace it,if you want customize.

#### Create a Custom Provider

# BLOB Storing: Creating a Custom Provider

This document explains how you can create a new storage provider for the BLOB storing system with an example.

> Read the [BLOB Storing document](Blob-Storing.md) to understand how to use the BLOB storing system. This document only covers how to create a new storage provider.

## Example Implementation

The first step is to create a class implements the `IBlobProvider` interface or inherit from the `BlobProviderBase` abstract class.

````csharp

using System.IO;

using System.Threading.Tasks;

using Volo.Abp.BlobStoring;

using Volo.Abp.DependencyInjection;

namespace AbpDemo

{

public class MyCustomBlobProvider : BlobProviderBase, ITransientDependency

{

public override Task SaveAsync(BlobProviderSaveArgs args)

{

//TODO...

}

public override Task<bool> DeleteAsync(BlobProviderDeleteArgs args)

{

//TODO...

}

public override Task<bool> ExistsAsync(BlobProviderExistsArgs args)

{

//TODO...

}

public override Task<Stream> GetOrNullAsync(BlobProviderGetArgs args)

{

//TODO...

}

}

}

````

\* `MyCustomBlobProvider` inherits from the `BlobProviderBase` and overrides the `abstract` methods. The actual implementation is up to you.

\* Implementing `ITransientDependency` registers this class to the [Dependency Injection](Dependency-Injection.md) system as a transient service.

> **\*\*Notice: Naming conventions are important\*\***. If your class name doesn't end with `BlobProvider`, you must manually register/expose your service for the `IBlobProvider`.

That's all. Now, you can configure containers (inside the `ConfigureServices` method of your [module](Module-Development-Basics.md)) to use the `MyCustomBlobProvider` class:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.ProviderType = typeof(MyCustomBlobProvider);

});

});

````

> See the [BLOB Storing document](Blob-Storing.md) if you want to configure a specific container.

### BlobContainerConfiguration Extension Method

If you want to provide a simpler configuration, create an extension method for the `BlobContainerConfiguration` class:

````csharp

public static class MyBlobContainerConfigurationExtensions

{

public static BlobContainerConfiguration UseMyCustomBlobProvider(

this BlobContainerConfiguration containerConfiguration)

{

containerConfiguration.ProviderType = typeof(MyCustomBlobProvider);

return containerConfiguration;

}

}

````

Then you can configure containers easier using the extension method:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseMyCustomBlobProvider();

});

});

````

### Extra Configuration Options

`BlobContainerConfiguration` allows to add/remove provider specific configuration objects. If your provider needs to additional configuration, you can create a wrapper class to the `BlobContainerConfiguration` for a type-safe configuration option:

````csharp

public class MyCustomBlobProviderConfiguration

{

public string MyOption1

{

get => \_containerConfiguration

.GetConfiguration<string>("MyCustomBlobProvider.MyOption1");

set => \_containerConfiguration

.SetConfiguration("MyCustomBlobProvider.MyOption1", value);

}

private readonly BlobContainerConfiguration \_containerConfiguration;

public MyCustomBlobProviderConfiguration(

BlobContainerConfiguration containerConfiguration)

{

\_containerConfiguration = containerConfiguration;

}

}

````

Then you can change the `MyBlobContainerConfigurationExtensions` class like that:

````csharp

public static class MyBlobContainerConfigurationExtensions

{

public static BlobContainerConfiguration UseMyCustomBlobProvider(

this BlobContainerConfiguration containerConfiguration,

Action<MyCustomBlobProviderConfiguration> configureAction)

{

containerConfiguration.ProviderType = typeof(MyCustomBlobProvider);

configureAction.Invoke(

new MyCustomBlobProviderConfiguration(containerConfiguration)

);

return containerConfiguration;

}

public static MyCustomBlobProviderConfiguration GetMyCustomBlobProviderConfiguration(

this BlobContainerConfiguration containerConfiguration)

{

return new MyCustomBlobProviderConfiguration(containerConfiguration);

}

}

````

\* Added an action parameter to the `UseMyCustomBlobProvider` method to allow developers to set the additional options.

\* Added a new `GetMyCustomBlobProviderConfiguration` method to be used inside `MyCustomBlobProvider` class to obtain the configured values.

Then anyone can set the `MyOption1` as shown below:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseMyCustomBlobProvider(provider =>

{

provider.MyOption1 = "my value";

});

});

});

````

Finally, you can access to the extra options using the `GetMyCustomBlobProviderConfiguration` method:

````csharp

public class MyCustomBlobProvider : BlobProviderBase, ITransientDependency

{

public override Task SaveAsync(BlobProviderSaveArgs args)

{

var config = args.Configuration.GetMyCustomBlobProviderConfiguration();

var value = config.MyOption1;

//...

}

}

````

## Contribute?

If you create a new provider and you think it can be useful for other developers, please consider to [contribute](Contribution/Index.md) to the ABP Framework on GitHub.

## Cancellation Token Provider

# Cancellation Token Provider

A `CancellationToken` enables cooperative cancellation between threads, thread pool work items, or `Task` objects. To handle the possible cancellation of the operation, ABP Framework provides `ICancellationTokenProvider` to obtain the `CancellationToken` itself from the source.

> To get more information about `CancellationToken`, see [Microsoft Documentation](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken).

## ICancellationTokenProvider

`ICancellationTokenProvider` is an abstraction to provide `CancellationToken` for different scenarios.

Generally, you should pass the `CancellationToken` as a parameter for your method to use it. With the `ICancellationTokenProvider` you don't need to pass `CancellationToken` for every method. `ICancellationTokenProvider` can be injected with the **\*\*dependency injection\*\*** and provides the token from it's source.

**\*\*Example:\*\***

```csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Threading;

namespace MyProject

{

public class MyService : ITransientDependency

{

private readonly ICancellationTokenProvider \_cancellationTokenProvider;

public MyService(ICancellationTokenProvider cancellationTokenProvider)

{

\_cancellationTokenProvider = cancellationTokenProvider;

}

public async Task DoItAsync()

{

while (\_cancellationTokenProvider.Token.IsCancellationRequested == false)

{

// ...

}

}

}

}

```

## Built-in providers

- `NullCancellationTokenProvider`

The `NullCancellationTokenProvider` is a built in provider and it supply always `CancellationToken.None`.

- `HttpContextCancellationTokenProvider`

The `HttpContextCancellationTokenProvider` is a built in default provider for ABP Web applications. It simply provides a `CancellationToken` that is source of the web request from the `HttpContext`.

## Implementing the ICancellationTokenProvider

You can easily create your CancellationTokenProvider by creating a class that implements the `ICancellationTokenProvider` interface, as shown below:

```csharp

using System.Threading;

namespace AbpDemo

{

public class MyCancellationTokenProvider : ICancellationTokenProvider

{

public CancellationToken Token { get; }

private MyCancellationTokenProvider()

{

}

}

}

```

## CSRF/XSRF & Anti Forgery

# CSRF/XSRF & Anti Forgery System

"*\*Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user’s web browser to perform an unwanted action on a trusted site for which the user is currently authenticated\**" ([OWASP](https://www.owasp.org/index.php/Cross-Site\_Request\_Forgery\_(CSRF)\_Prevention\_Cheat\_Sheet)).

**\*\*ABP Framework completely automates CSRF preventing\*\*** and works out of the box without any configuration. Read this documentation only if you want to understand it better or need to customize.

## The Problem

ASP.NET Core [provides infrastructure](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery) to prevent CSRF attacks by providing a system to **\*\*generate\*\*** and **\*\*validate antiforgery tokens\*\***. However, the standard implementation has a few drawbacks;

Antiforgery token validation is only **\*\*enabled for razor pages by default\*\*** and not enabled for **\*\*HTTP APIs\*\***. You need to enable it yourself for the Controllers. You can use the `[ValidateAntiForgeryToken]` attribute for a specific API Controller/Action or the `[AutoValidateAntiforgeryToken]` attribute to prevent attacks globally.

Once you enable it;

\* You need to manually add an HTTP header, named `RequestVerificationToken` to every **\*\*AJAX request\*\*** made in your application. You should care about obtaining the token, saving in the client side and adding to the HTTP header on every HTTP request.

\* All your clients, including **\*\*non-browser clients\*\***, should care about obtaining and sending the antiforgery token in every request. In fact, non-browser clients has no CSRF risk and should not care about this.

Especially, the second point is a pain for your clients and unnecessarily consumes your server resources.

> You can read more about the ASP.NET Core antiforgery system in its own [documentation](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery).

## The Solution

ABP Framework provides `[AbpValidateAntiForgeryToken]` and `[AbpAutoValidateAntiforgeryToken]` attributes, just like the attributes explained above. `[AbpAutoValidateAntiforgeryToken]` is already added to the global filters, so you should do nothing to enable it for your application.

ABP Framework also automates the following infrastructure;

\* Server side sets a **\*\*special cookie\*\***, named `XSRF-TOKEN` by default, that is used make the antiforgery token value available to the browser. This is **\*\*done automatically\*\*** (by the [application configuration](API/Application-Configuration.md) endpoint). Nothing to do in the client side.

\* In the client side, it reads the token from the cookie and sends it in the **\*\*HTTP header\*\*** (named `RequestVerificationToken` by default). This is implemented for all the supported UI types.

\* Server side validates the antiforgery token **\*\*only for same and cross site requests\*\*** made by the browser. It bypasses the validation for non-browser clients.

That's all. The systems works smoothly.

## Configuration / Customization

### AbpAntiForgeryOptions

`AbpAntiForgeryOptions` is the main [options class](Options.md) to configure the ABP Antiforgery system. It has the following properties;

\* `TokenCookie`: Can be used to configure the cookie details. This cookie is used to store the antiforgery token value in the client side, so clients can read it and sends the value as the HTTP header. Default cookie name is `XSRF-TOKEN`, expiration time is 10 years (yes, ten years! It should be a value longer than the authentication cookie max life time, for the security).

\* `AuthCookieSchemaName`: The name of the authentication cookie used by your application. Default value is `Identity.Application` (which becomes `AspNetCore.Identity.Application` on runtime). The default value properly works with the ABP startup templates. **\*\*If you change the authentication cookie name, you also must change this.\*\***

\* `AutoValidate`: The single point to enable/disable the ABP automatic antiforgery validation system. Default value is `true`.

\* `AutoValidateFilter`: A predicate that gets a type and returns a boolean. ABP uses this predicate to check a controller type. If it returns false for a controller type, the controller is excluded from the automatic antiforgery token validation.

\* `AutoValidateIgnoredHttpMethods`: A list of HTTP Methods to ignore on automatic antiforgery validation. Default value: "GET", "HEAD", "TRACE", "OPTIONS". These HTTP Methods are safe to skip antiforgery validation since they don't change the application state.

If you need to change these options, do it in the `ConfigureServices` method of your [module](Module-Development-Basics.md).

**\*\*Example: Configuring the AbpAntiForgeryOptions\*\***

```csharp

Configure<AbpAntiForgeryOptions>(options =>

{

options.TokenCookie.Expiration = TimeSpan.FromDays(365);

options.AutoValidateIgnoredHttpMethods.Remove("GET");

options.AutoValidateFilter =

type => !type.Namespace.StartsWith("MyProject.MyIgnoredNamespace");

});

```

This configuration;

\* Sets the antiforgery token expiration time to ~1 year.

\* Enables antiforgery token validation for GET requests too.

\* Ignores the controller types in the specified namespace.

### AntiforgeryOptions

`AntiforgeryOptions` is the standard [options class](Options.md) of the ASP.NET Core. **\*\*You can find all the information about this class in its [**own documentation**](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery)\*\***.

`HeaderName` option is especially important for the ABP Framework point of view. Default value of this value is `RequestVerificationToken` and the clients uses this name while sending the token value in the header. So, if you change this option, you should also arrange your clients to align the change. If you don't have a good reason, leave it as default.

### AbpValidateAntiForgeryToken Attribute

If you disable the automatic validation or want to perform the validation for an endpoint that is not validated by default (for example, an endpoint with HTTP GET Method), you can use the `[AbpValidateAntiForgeryToken]` attribute for a **\*\*controller type or method\*\*** (action).

**\*\*Example: Add** `[AbpValidateAntiForgeryToken]` **to a HTTP GET method\*\***

```csharp

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.AntiForgery;

namespace MyCompanyName.MyProjectName.Controllers

{

[Route("api/products")]

public class ProductController : AbpController

{

[HttpGet]

[AbpValidateAntiForgeryToken]

public async Task GetAsync()

{

//TODO: ...

}

}

}

```

### Angular UI

Angular supports CSRF Token out of box, but the token header name is `X-XSRF-TOKEN`. Since ABP Framework follows the ASP.NET Core conventions, it changes this value to `RequestVerificationToken` in the core package.

You don't need to make anything unless you need to change the `AntiforgeryOptions.HeaderName` as explained before. If you change it, remember to change the header name for the Angular application too. To do that, add an import declaration for the `HttpClientXsrfModule` into your root module.

**\*\*Example: Change the header name to *\*MyCustomHeaderName\**\*\***

```typescript

@NgModule({

// ...

imports: [

//...

HttpClientXsrfModule.withOptions({

cookieName: 'XSRF-TOKEN',

headerName: 'MyCustomHeaderName'

})

],

})

export class AppModule {}

```

**\*\*Note:\*\*** XSRF-TOKEN is only valid if both frontend application and APIs run on the same domain. Therefore, when you make a request, you should use a relative path.

For example, let's say your APIs is hosted at `https://testdomain.com/ws`

and your angular application is hosted at `https://testdomain.com/admin`

So if your API request should look like this `https://testdomain.com/ws/api/identity/users`

your `environment.prod.ts` has to be as follows:

```typescript

export const environment = {

production: true,

// ....

apis: {

default: {

url: '/ws', // <- just use the context root here

// ...

},

},

} as Config.Environment;

```

Let's talk about why.

First, take a look at [Angular's code](https://github.com/angular/angular/blob/master/packages/common/http/src/xsrf.ts#L81)

It does not intercept any request that starts with `http://` or `https://`. There is a good reason for that. Any cross-site request does not need this token for security. This verification is only valid if the request is made to the same domain from which the web page is served. So, simply put, if you serve everything from a single domain, you just use a relative path.

If you serve your APIs from the root, i.e. no context root (https://testdomain.com/api/identity/users), leave `url` empty as follows:

```typescript

export const environment = {

production: true,

// ....

apis: {

default: {

url: '', // <- should be empty string, not '/'

// ...

},

},

} as Config.Environment;

```

## Concurrency Check

## Concurrency Check

### Introduction

Concurrency Check (also known as **\*\*Concurrency Control\*\***) refers to specific mechanisms used to ensure data consistency in the presence of concurrent changes (multiple processes, users access or change the same data in a database at the same time).

There are two commonly used concurrency control mechanisms/approaches:

\* **\*\*Optimistic Concurrency Control\*\***: Optimistic Concurrency Control allows multiple users to attempt to **\*\*update\*\*** the same record without informing the users that others are also attempting to **\*\*update\*\*** it.

\* If a user successfully updates the record, the other users need to get the latest changes for the current record to be able to make changes.

\* ABP's concurrency check system uses the **\*\*Optimistic Concurrency Control\*\***.

\* **\*\*Pessimistic Concurrency Control\*\***: Pessimistic Concurrency Control prevents simultaneous updates to records and uses a locking mechanism. For more information please see [here](https://www.martinfowler.com/eaaCatalog/pessimisticOfflineLock.html).

### Usage

#### `IHasConcurrencyStamp` Interface

To enable **\*\*concurrency control\*\*** to your entity class, you should implement the `IHasConcurrencyStamp` interface, directly or indirectly.

```csharp

public interface IHasConcurrencyStamp

{

public string ConcurrencyStamp { get; set; }

}

```

\* It is the base interface for **\*\*concurrency control\*\*** and only has a simple property named `ConcurrencyStamp`.

\* While a new record is **\*\*creating\*\***, if the entity implements the `IHasConcurrencyStamp` interface, ABP Framework automatically sets a unique value to the **\*\*ConcurrencyStamp\*\*** property.

\* While a record is **\*\*updating\*\***, ABP Framework compares the **\*\*ConcurrencyStamp\*\*** property of the entity with the provided **\*\*ConcurrencyStamp\*\*** value by the user and if the values match, it automatically updates the **\*\*ConcurrencyStamp\*\*** property with the new unique value. If there is a mismatch, `AbpDbConcurrencyException` is thrown.

**\*\*Example: Applying Concurrency Control for the Book Entity\*\***

Implement the `IHasConcurrencyStamp` interface for your entity:

```csharp

public class Book : Entity<Guid>, IHasConcurrencyStamp

{

public string ConcurrencyStamp { get; set; }

//...

}

```

Also, implement your output and update the DTO classes from the `IHasConcurrencyStamp` interface:

```csharp

public class BookDto : EntityDto<Guid>, IHasConcurrencyStamp

{

//...

public string ConcurrencyStamp { get; set; }

}

public class UpdateBookDto : IHasConcurrencyStamp

{

//...

public string ConcurrencyStamp { get; set; }

}

```

Set the **\*\*ConcurrencyStamp\*\*** input value to the entity in the **\*\*UpdateAsync\*\*** method of your application service as below:

```csharp

public class BookAppService : ApplicationService, IBookAppService

{

//...

public virtual async Task<BookDto> UpdateAsync(Guid id, UpdateBookDto input)

{

var book = await BookRepository.GetAsync(id);

book.ConcurrencyStamp = input.ConcurrencyStamp;

//set other input values to the entity ...

await BookRepository.UpdateAsync(book);

}

}

```

\* After that, when multiple users try to update the same record at the same time, the concurrency stamp mismatch occurs and `AbpDbConcurrencyException` is thrown.

#### Base Classes

[Aggregate Root](./Entities.md#aggregateroot-class) entity classes already implement the `IHasConcurrencyStamp` interface. So, if you are deriving from one of these base classes, you don't need to manually implement the `IHasConcurrencyStamp` interface:

- `AggregateRoot`, `AggregateRoot<TKey>`

- `CreationAuditedAggregateRoot`, `CreationAuditedAggregateRoot<TKey>`

- `AuditedAggregateRoot`, `AuditedAggregateRoot<TKey>`

- `FullAuditedAggregateRoot`, `FullAuditedAggregateRoot<TKey>`

**\*\*Example: Applying Concurrency Control for the Book Entity\*\***

You can inherit your entity from one of [the base classes](#base-classes):

```csharp

public class Book : FullAuditedAggregateRoot<Guid>

{

//...

}

```

Then, you can implement your output and update the DTO classes from the `IHasConcurrencyStamp` interface:

```csharp

public class BookDto : EntityDto<Guid>, IHasConcurrencyStamp

{

//...

public string ConcurrencyStamp { get; set; }

}

public class UpdateBookDto : IHasConcurrencyStamp

{

//...

public string ConcurrencyStamp { get; set; }

}

```

Set the **\*\*ConcurrencyStamp\*\*** input value to the entity in the **\*\*UpdateAsync\*\*** method of your application service as below:

```csharp

public class BookAppService : ApplicationService, IBookAppService

{

//...

public virtual async Task<BookDto> UpdateAsync(Guid id, UpdateBookDto input)

{

var book = await BookRepository.GetAsync(id);

book.ConcurrencyStamp = input.ConcurrencyStamp;

//set other input values to the entity ...

await BookRepository.UpdateAsync(book);

}

}

```

After that, when multiple users try to update the same record at the same time, the concurrency stamp mismatch occurs and `AbpDbConcurrencyException` is thrown. You can either handle the exception manually or let the ABP Framework handle it for you.

ABP Framework shows a user-friendly error message as in the image below, if you don't handle the exception manually.

![Optimistic Concurrency](./images/optimistic-concurrency.png)

## Current User

# Current User

It is very common to retrieve the information about the logged in user in a web application. The current user is the active user related to the current request in a web application.

## ICurrentUser

`ICurrentUser` is the main service to get info about the current active user.

Example: [Injecting](Dependency-Injection.md) the `ICurrentUser` into a service:

````csharp

using System;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Users;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly ICurrentUser \_currentUser;

public MyService(ICurrentUser currentUser)

{

\_currentUser = currentUser;

}

public void Foo()

{

Guid? userId = \_currentUser.Id;

}

}

}

````

Common base classes have already injected this service as a base property. For example, you can directly use the `CurrentUser` property in an [application service](Application-Services.md):

````csharp

using System;

using Volo.Abp.Application.Services;

namespace AbpDemo

{

public class MyAppService : ApplicationService

{

public void Foo()

{

Guid? userId = CurrentUser.Id;

}

}

}

````

### Properties

Here are the fundamental properties of the `ICurrentUser` interface:

\* **\*\*IsAuthenticated\*\*** (bool): Returns `true` if the current user has logged in (authenticated). If the user has not logged in then `Id` and `UserName` returns `null`.

\* **\*\*Id\*\*** (Guid?): Id of the current user. Returns `null`, if the current user has not logged in.

\* **\*\*UserName\*\*** (string): User name of the current user. Returns `null`, if the current user has not logged in.

\* **\*\*TenantId\*\*** (Guid?): Tenant Id of the current user, which can be useful for a [multi-tenant](Multi-Tenancy.md) application. Returns `null`, if the current user is not assigned to a tenant.

\* **\*\*Email\*\*** (string): Email address of the current user.Returns `null`, if the current user has not logged in or not set an email address.

\* **\*\*EmailVerified\*\*** (bool): Returns `true`, if the email address of the current user has been verified.

\* **\*\*PhoneNumber\*\*** (string): Phone number of the current user. Returns `null`, if the current user has not logged in or not set a phone number.

\* **\*\*PhoneNumberVerified\*\*** (bool): Returns `true`, if the phone number of the current user has been verified.

\* **\*\*Roles\*\*** (string[]): Roles of the current user. Returns a string array of the role names of the current user.

### Methods

`ICurrentUser` is implemented on the `ICurrentPrincipalAccessor` (see the section below) and works with the claims. So, all of the above properties are actually retrieved from the claims of the current authenticated user.

`ICurrentUser` has some methods to directly work with the claims, if you have custom claims or get other non-common claim types.

\* **\*\*FindClaim\*\***: Gets a claim with the given name. Returns `null` if not found.

\* **\*\*FindClaims\*\***: Gets all the claims with the given name (it is allowed to have multiple claim values with the same name).

\* **\*\*GetAllClaims\*\***: Gets all the claims.

\* **\*\*IsInRole\*\***: A shortcut method to check if the current user is in the specified role.

Beside these standard methods, there are some extension methods:

\* **\*\*FindClaimValue\*\***: Gets the value of the claim with the given name, or `null` if not found. It has a generic overload that also casts the value to a specific type.

\* **\*\*GetId\*\***: Returns `Id` of the current user. If the current user has not logged in, it throws an exception (instead of returning `null`) . Use this only if you are sure that the user has already authenticated in your code context.

### Authentication & Authorization

`ICurrentUser` works independently of how the user is authenticated or authorized. It seamlessly works with any authentication system that works with the current principal (see the section below).

## ICurrentPrincipalAccessor

`ICurrentPrincipalAccessor` is the service that should be used (by the ABP Framework and your application code) whenever the current principal of the current user is needed.

For a web application, it gets the `User` property of the current `HttpContext`. For a non-web application, it returns the `Thread.CurrentPrincipal`.

> You generally don't need to use this low level `ICurrentPrincipalAccessor` service and just directly work with the `ICurrentUser` explained above.

### Basic Usage

You can inject `ICurrentPrincipalAccessor` and use the `Principal` property to the the current principal:

````csharp

public class MyService : ITransientDependency

{

private readonly ICurrentPrincipalAccessor \_currentPrincipalAccessor;

public MyService(ICurrentPrincipalAccessor currentPrincipalAccessor)

{

\_currentPrincipalAccessor = currentPrincipalAccessor;

}

public void Foo()

{

var allClaims = \_currentPrincipalAccessor.Principal.Claims.ToList();

//...

}

}

````

### Changing the Current Principal

Current principal is not something you want to set or change, except at some advanced scenarios. If you need it, use the `Change` method of the `ICurrentPrincipalAccessor`. It takes a `ClaimsPrincipal` object and makes it "current" for a scope.

Example:

````csharp

public class MyAppService : ApplicationService

{

private readonly ICurrentPrincipalAccessor \_currentPrincipalAccessor;

public MyAppService(ICurrentPrincipalAccessor currentPrincipalAccessor)

{

\_currentPrincipalAccessor = currentPrincipalAccessor;

}

public void Foo()

{

var newPrincipal = new ClaimsPrincipal(

new ClaimsIdentity(

new Claim[]

{

new Claim(AbpClaimTypes.UserId, Guid.NewGuid().ToString()),

new Claim(AbpClaimTypes.UserName, "john"),

new Claim("MyCustomCliam", "42")

}

)

);

using (\_currentPrincipalAccessor.Change(newPrincipal))

{

var userName = CurrentUser.UserName; //returns "john"

//...

}

}

}

````

Use the `Change` method always in a `using` statement, so it will be restored to the original value after the `using` scope ends.

This can be a way to simulate a user login for a scope of the application code, however try to use it carefully.

## AbpClaimTypes

`AbpClaimTypes` is a static class that defines the names of the standard claims and used by the ABP Framework.

\* Default values for the `UserName`, `UserId`, `Role` and `Email` properties are set from the [System.Security.Claims.ClaimTypes](https://docs.microsoft.com/en-us/dotnet/api/system.security.claims.claimtypes) class, but you can change them.

\* Other properties, like `EmailVerified`, `PhoneNumber`, `TenantId`... are defined by the ABP Framework by following the standard names wherever possible.

It is suggested to use properties of this class instead of magic strings for claim names.

## Data Filtering

# Data Filtering

[Volo.Abp.Data](https://www.nuget.org/packages/Volo.Abp.Data) package defines services to automatically filter data on querying from a database.

## Pre-Defined Filters

ABP defines some filters out of the box.

### ISoftDelete

Used to mark an [entity](Entities.md) as deleted instead of actually deleting it. Implement the `ISoftDelete` interface to make your entity "soft delete".

Example:

````csharp

using System;

using Volo.Abp;

using Volo.Abp.Domain.Entities;

namespace Acme.BookStore

{

public class Book : AggregateRoot<Guid>, ISoftDelete

{

public string Name { get; set; }

public bool IsDeleted { get; set; } //Defined by ISoftDelete

}

}

````

`ISoftDelete` defines the `IsDeleted` property. When you delete a book using [repositories](Repositories.md), ABP automatically sets `IsDeleted` to true and protects it from actual deletion (you can also manually set the `IsDeleted` property to true if you need). In addition, it **\*\*automatically filters deleted entities\*\*** when you query the database.

> `ISoftDelete` filter is enabled by default and you can not get deleted entities from database unless you explicitly disable it. See the `IDataFilter` service below.

> Soft-delete entities can be hard-deleted when you use `HardDeleteAsync` method on the repositories.

### IMultiTenant

[Multi-tenancy](Multi-Tenancy.md) is an efficient way of creating SaaS applications. Once you create a multi-tenant application, you typically want to isolate data between tenants. Implement `IMultiTenant` interface to make your entity "multi-tenant aware".

Example:

````csharp

using System;

using Volo.Abp;

using Volo.Abp.Domain.Entities;

using Volo.Abp.MultiTenancy;

namespace Acme.BookStore

{

public class Book : AggregateRoot<Guid>, ISoftDelete, IMultiTenant

{

public string Name { get; set; }

public bool IsDeleted { get; set; } //Defined by ISoftDelete

public Guid? TenantId { get; set; } //Defined by IMultiTenant

}

}

````

`IMultiTenant` interface defines the `TenantId` property which is then used to automatically filter the entities for the current tenant. See the [Multi-tenancy](Multi-Tenancy.md) document for more.

## IDataFilter Service: Enable/Disable Data Filters

You can control the filters using `IDataFilter` service.

Example:

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore

{

public class MyBookService : ITransientDependency

{

private readonly IDataFilter \_dataFilter;

private readonly IRepository<Book, Guid> \_bookRepository;

public MyBookService(

IDataFilter dataFilter,

IRepository<Book, Guid> bookRepository)

{

\_dataFilter = dataFilter;

\_bookRepository = bookRepository;

}

public async Task<List<Book>> GetAllBooksIncludingDeletedAsync()

{

//Temporary disable the ISoftDelete filter

using (\_dataFilter.Disable<ISoftDelete>())

{

return await \_bookRepository.GetListAsync();

}

}

}

}

````

\* [Inject](Dependency-Injection.md) the `IDataFilter` service to your class.

\* Use the `Disable` method within a `using` statement to create a code block where the `ISoftDelete` filter is disabled inside it.

In addition to the `Disable<T>()` method;

\* `IDataFilter.Enable<T>()` method can be used to enable a filter. `Enable` and `Disable` methods can be used in a **\*\*nested\*\*** way to define inner scopes.

\* `IDataFilter.IsEnabled<T>()` can be used to check whether a filter is currently enabled or not.

> Always use the `Disable` and `Enable` methods it inside a `using` block to guarantee that the filter is reset to its previous state.

### The Generic IDataFilter Service

`IDataFilter` service has a generic version, `IDataFilter<TFilter>` that injects a more restricted and explicit data filter based on the filter type.

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore

{

public class MyBookService : ITransientDependency

{

private readonly IDataFilter<ISoftDelete> \_softDeleteFilter;

private readonly IRepository<Book, Guid> \_bookRepository;

public MyBookService(

IDataFilter<ISoftDelete> softDeleteFilter,

IRepository<Book, Guid> bookRepository)

{

\_softDeleteFilter = softDeleteFilter;

\_bookRepository = bookRepository;

}

public async Task<List<Book>> GetAllBooksIncludingDeletedAsync()

{

//Temporary disable the ISoftDelete filter

using (\_softDeleteFilter.Disable())

{

return await \_bookRepository.GetListAsync();

}

}

}

}

````

\* This usage determines the filter type while injecting the `IDataFilter<T>` service.

\* In this case you can use the `Disable()` and `Enable()` methods without specifying the filter type.

## AbpDataFilterOptions

`AbpDataFilterOptions` can be used to [set options](Options.md) for the data filter system.

The example code below disables the `ISoftDelete` filter by default which will cause to include deleted entities when you query the database unless you explicitly enable the filter:

````csharp

Configure<AbpDataFilterOptions>(options =>

{

options.DefaultStates[typeof(ISoftDelete)] = new DataFilterState(isEnabled: false);

});

````

> Carefully change defaults for global filters, especially if you are using a pre-built module which might be developed assuming the soft delete filter is turned on by default. But you can do it for your own defined filters safely.

## Defining Custom Filters

Defining and implementing a new filter highly depends on the database provider. ABP implements all pre-defined filters for all database providers.

When you need it, start by defining an interface (like `ISoftDelete` and `IMultiTenant`) for your filter and implement it for your entities.

Example:

````csharp

public interface IIsActive

{

bool IsActive { get; }

}

````

Such an `IIsActive` interface can be used to filter active/passive data and can be easily implemented by any [entity](Entities.md):

````csharp

public class Book : AggregateRoot<Guid>, IIsActive

{

public string Name { get; set; }

public bool IsActive { get; set; } //Defined by IIsActive

}

````

### EntityFramework Core

ABP uses [EF Core's Global Query Filters](https://docs.microsoft.com/en-us/ef/core/querying/filters) system for the [EF Core Integration](Entity-Framework-Core.md). So, it is well integrated to EF Core and works as expected even if you directly work with `DbContext`.

Best way to implement a custom filter is to override `ShouldFilterEntity` and `CreateFilterExpression` method for your `DbContext`. Example:

````csharp

protected bool IsActiveFilterEnabled => DataFilter?.IsEnabled<IIsActive>() ?? false;

protected override bool ShouldFilterEntity<TEntity>(IMutableEntityType entityType)

{

if (typeof(IIsActive).IsAssignableFrom(typeof(TEntity)))

{

return true;

}

return base.ShouldFilterEntity<TEntity>(entityType);

}

protected override Expression<Func<TEntity, bool>> CreateFilterExpression<TEntity>()

{

var expression = base.CreateFilterExpression<TEntity>();

if (typeof(IIsActive).IsAssignableFrom(typeof(TEntity)))

{

Expression<Func<TEntity, bool>> isActiveFilter =

e => !IsActiveFilterEnabled || EF.Property<bool>(e, "IsActive");

expression = expression == null

? isActiveFilter

: CombineExpressions(expression, isActiveFilter);

}

return expression;

}

````

\* Added a `IsActiveFilterEnabled` property to check if `IIsActive` is enabled or not. It internally uses the `IDataFilter` service introduced before.

\* Overrided the `ShouldFilterEntity` and `CreateFilterExpression` methods, checked if given entity implements the `IIsActive` interface and combines the expressions if necessary.

In addition you can also use `HasAbpQueryFilter` to set a filter for an entity. It will combine your filter with ABP EF Core builtin global query filters.

````csharp

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<MyEntity>(b =>

{

b.HasAbpQueryFilter(e => e.Name.StartsWith("abp"));

});

}

````

### MongoDB

ABP abstracts the `IMongoDbRepositoryFilterer` interface to implement data filtering for the [MongoDB Integration](MongoDB.md), it works only if you use the repositories properly. Otherwise, you should manually filter the data.

Currently, the best way to implement a data filter for the MongoDB integration is to create a derived class of `MongoDbRepositoryFilterer` and override `FilterQueryable`. Example:

````csharp

[ExposeServices(typeof(IMongoDbRepositoryFilterer<Book, Guid>))]

public class BookMongoDbRepositoryFilterer : MongoDbRepositoryFilterer<Book, Guid> , ITransientDependency

{

public BookMongoDbRepositoryFilterer(

IDataFilter dataFilter,

ICurrentTenant currentTenant) :

base(dataFilter, currentTenant)

{

}

public override TQueryable FilterQueryable<TQueryable>(TQueryable query)

{

if (DataFilter.IsEnabled<IIsActive>())

{

return (TQueryable)query.Where(x => x.IsActive);

}

return base.FilterQueryable(query);

}

}

````

This example implements it only for the `Book` entity. If you want to implement for all entities (those implement the `IIsActive` interface), create your own custom MongoDB repository filterer base class and override the `AddGlobalFilters` as shown below:

````csharp

public abstract class MyMongoRepository<TMongoDbContext, TEntity, TKey> : MongoDbRepository<TMongoDbContext, TEntity, TKey>

where TMongoDbContext : IAbpMongoDbContext

where TEntity : class, IEntity<TKey>

{

protected MyMongoRepository(IMongoDbContextProvider<TMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

protected override void AddGlobalFilters(List<FilterDefinition<TEntity>> filters)

{

base.AddGlobalFilters(filters);

if (typeof(IIsActive).IsAssignableFrom(typeof(TEntity))

&& DataFilter.IsEnabled<IIsActive>())

{

filters.Add(Builders<TEntity>.Filter.Eq(e => ((IIsActive)e).IsActive, true));

}

}

}

public class MyMongoDbModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//.......

context.Services

.Replace(ServiceDescriptor.Transient(typeof(IMongoDbRepositoryFilterer<,>),typeof(MyMongoDbRepositoryFilterer<,>)));

}

}

````

## Data Seeding

# Data Seeding

## Introduction

Some applications (or modules) using a database may need to have some **\*\*initial data\*\*** to be able to properly start and run. For example, an **\*\*admin user\*\*** & roles must be available at the beginning. Otherwise you can not **\*\*login\*\*** to the application to create new users and roles.

Data seeding is also useful for [testing](Testing.md) purpose, so your automatic tests can assume some initial data available in the database.

### Why a Data Seed System?

While EF Core Data Seeding system provides a way, it is very limited and doesn't cover production scenarios. Also, it is only for EF Core.

ABP Framework provides a data seed system that is;

\* **\*\*Modular\*\***: Any [module](Module-Development-Basics.md) can silently contribute to the data seeding process without knowing and effecting each other. In this way, a module seeds its own initial data.

\* **\*\*Database Independent\*\***: It is not only for EF Core, it also works for other database providers (like [MongoDB](MongoDB.md)).

\* **\*\*Production Ready\*\***: It solves the problems on production environments. See the "*\*On Production\**" section below.

\* **\*\*Dependency Injection\*\***: It takes the full advantage of dependency injection, so you can use any internal or external service while seeding the initial data. Actually, you can do much more than data seeding.

## IDataSeedContributor

`IDataSeedContributor` is the interface that should be implemented in order to seed data to the database.

**\*\*Example: Seed one initial book to the database if there is no book\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Guids;

namespace Acme.BookStore

{

public class BookStoreDataSeedContributor

: IDataSeedContributor, ITransientDependency

{

private readonly IRepository<Book, Guid> \_bookRepository;

private readonly IGuidGenerator \_guidGenerator;

private readonly ICurrentTenant \_currentTenant;

public BookStoreDataSeedContributor(

IRepository<Book, Guid> bookRepository,

IGuidGenerator guidGenerator,

ICurrentTenant currentTenant)

{

\_bookRepository = bookRepository;

\_guidGenerator = guidGenerator;

\_currentTenant = currentTenant;

}

public async Task SeedAsync(DataSeedContext context)

{

using (\_currentTenant.Change(context?.TenantId))

{

if (await \_bookRepository.GetCountAsync() > 0)

{

return;

}

var book = new Book(

id: \_guidGenerator.Create(),

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

type: BookType.ScienceFiction,

publishDate: new DateTime(1979, 10, 12),

price: 42

);

await \_bookRepository.InsertAsync(book);

}

}

}

}

````

\* `IDataSeedContributor` defines the `SeedAsync` method to execute the **\*\*data seed logic\*\***.

\* It is typical to **\*\*check database\*\*** if the seeding data is already present.

\* You can **\*\*inject\*\*** service and perform any logic needed to seed the data.

> Data seed contributors are automatically discovered by the ABP Framework and executed as a part of the data seed process.

### DataSeedContext

`DataSeedContext` contains `TenantId` if your application is [multi-tenant](Multi-Tenancy.md), so you can use this value while inserting data or performing custom logic based on the tenant.

`DataSeedContext` also contains name-value style configuration parameters for passing to the seeder contributors from the `IDataSeeder`.

## Modularity

An application can have multiple data seed contributor (`IDataSeedContributor`) class. So, any reusable module can also implement this interface to seed its own initial data.

For example, the [Identity Module](Modules/Identity.md) has a data seed contributor that creates an admin role and admin user and assign all the permissions.

## IDataSeeder

> You typically never need to directly use the `IDataSeeder` service since it is already done if you've started with the [application startup template](Startup-Templates/Application.md). But its suggested to read it to understand the design behind the data seed system.

`IDataSeeder` is the main service that is used to seed initial data. It is pretty easy to use;

````csharp

public class MyService : ITransientDependency

{

private readonly IDataSeeder \_dataSeeder;

public MyService(IDataSeeder dataSeeder)

{

\_dataSeeder = dataSeeder;

}

public async Task FooAsync()

{

await \_dataSeeder.SeedAsync();

}

}

````

You can [inject](Dependency-Injection.md) the `IDataSeeder` and use it to seed the initial data when you need. It internally calls all the `IDataSeedContributor` implementations to complete the data seeding.

It is possible to send named configuration parameters to the `SeedAsync` method as shown below:

````csharp

await \_dataSeeder.SeedAsync(

new DataSeedContext()

.WithProperty("MyProperty1", "MyValue1")

.WithProperty("MyProperty2", 42)

);

````

Then the data seed contributors can access to these properties via the `DataSeedContext` explained before.

If a module needs to a parameter, it should be declared on the [module documentation](Modules/Index.md). For example, the [Identity Module](Modules/Identity.md) can use `AdminEmail` and `AdminPassword` parameters if you provide (otherwise uses the default values).

### Separate Unit Of Works

The default seed will be in a unit of work and may use transactions. If there are multiple `IDataSeedContributor` or too much data written, it may cause a database timeout error.

We provide an extension method of `SeedInSeparateUowAsync` for the `IDataSeeder` service to create a separate unit of work for each `IDataSeedContributor`.

````csharp

public static Task SeedInSeparateUowAsync(this IDataSeeder seeder, Guid? tenantId = null, AbpUnitOfWorkOptions options = null, bool requiresNew = false)

````

### Where & How to Seed Data?

It is important to understand where & how to execute the `IDataSeeder.SeedAsync()`?

#### On Production

The [application startup template](Startup-Templates/Application.md) comes with a *\*YourProjectName\**\*\*.DbMigrator\*\* project (Acme.BookStore.DbMigrator on the picture below), which is a **\*\*console application\*\*** that is responsible to **\*\*migrate\*\*** the database schema (for relational databases) and **\*\*seed\*\*** the initial data:

![bookstore-visual-studio-solution-v3](images/bookstore-visual-studio-solution-v3.png)

This console application is properly configured for you. It even supports **\*\*multi-tenant\*\*** scenarios where each tenant has its own database (migrates & seeds all necessary databases).

It is expected to run this DbMigrator application whenever you **\*\*deploy a new version\*\*** of your solution to the server. It will migrate your **\*\*database schema\*\*** (create new tables/fields... etc.) and **\*\*seed new initial data\*\*** needed to properly run the new version of your solution. Then you can deploy/start your actual application.

Even if you are using MongoDB or another NoSQL database (that doesn't need to schema migrations), it is recommended to use the DbMigrator application to seed your data or perform your data migration.

Having such a separate console application has several advantages;

\* You can **\*\*run it before\*\*** updating your application, so your application will run on the ready database.

\* Your application **\*\*starts faster\*\*** compared to if it seeds the initial data itself.

\* Your application can properly run on a **\*\*clustered environment\*\*** (where multiple instances of your application run concurrently). If you seed data on application startup you would have conflicts in this case.

#### On Development

We suggest the same way on development. Run the DbMigrator console application whenever you [create a database migration](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/migrations/) (using EF Core `Add-Migration` command, for example) or change the data seed code (will be explained later).

> You can continue to use the standard `Update-Database` command for EF Core, but it will not seed if you've created a new seed data.

#### On Testing

You probably want to seed the data also for automated [testing](Testing.md), so want to use the `IDataSeeder.SeedAsync()`. In the [application startup template](Startup-Templates/Application.md), it is done in the [OnApplicationInitialization](Module-Development-Basics.md) method of the *\*YourProjectName\**TestBaseModule class of the TestBase project.

In addition to the standard seed data (that is also used on production), you may want to seed additional data unique to the automated tests. If so, you can create a new data seed contributor in the test project to have more data to work on.

## Distributed Locking

## Email Sending

### Email Sending System

# Email Sending

ABP Framework provides various services, settings and integrations for sending emails;

\* Provides `IEmailSender` service that is used to send emails.

\* Defines [settings](Settings.md) to configure email sending.

\* Integrates to the [background job system](Background-Jobs.md) to send emails via background jobs.

\* Provides [MailKit integration](MailKit.md) package.

## Installation

> This package is already installed if you are using the [application startup template](Startup-Templates/Application.md).

It is suggested to use the [ABP CLI](CLI.md) to install this package. Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.Emailing

````

If you haven't done it yet, you first need to install the ABP CLI. For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Emailing).

## Sending Emails

### IEmailSender

[Inject](Dependency-Injection.md) the `IEmailSender` into any service and use the `SendAsync` method to send emails.

**\*\*Example\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

namespace MyProject

{

public class MyService : ITransientDependency

{

private readonly IEmailSender \_emailSender;

public MyService(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public async Task DoItAsync()

{

await \_emailSender.SendAsync(

"target@domain.com", // target email address

"Email subject", // subject

"This is email body..." // email body

);

}

}

}

````

`SendAsync` method has overloads to supply more parameters like;

\* **\*\*from\*\***: You can set this as the first argument to set a sender email address. If not provided, the default sender address is used (see the email settings below).

\* **\*\*isBodyHtml\*\***: Indicates whether the email body may contain HTML tags. **\*\*Default: true\*\***.

> `IEmailSender` is the suggested way to send emails, since it makes your code provider independent.

#### MailMessage

In addition to primitive parameters, you can pass a **\*\*standard** `MailMessage` **object\*\*** ([see](https://docs.microsoft.com/en-us/dotnet/api/system.net.mail.mailmessage)) to the `SendAsync` method to set more options, like adding attachments.

### ISmtpEmailSender

Sending emails is implemented by the standard `SmtpClient` class ([see](https://docs.microsoft.com/en-us/dotnet/api/system.net.mail.smtpclient)) by default. The implementation class is the `SmtpEmailSender`. This class also expose the `ISmtpEmailSender` service (in addition to the `IEmailSender`).

Most of the time you want to directly use the `IEmailSender` to make your code provider independent. However, if you want to create an `SmtpClient` object with the same email settings, you can inject the `ISmtpEmailSender` and use its `BuildClientAsync` method to obtain a `SmtpClient` object and send the email yourself.

## Queueing Emails / Background Jobs

`IEmailSender` has a `QueueAsync` method that can be used to add emails to the background job queue to send them in a background thread. In this way, you don't take time of the user by waiting to send the email. `QueueAsync` method gets the same arguments with the `SendAsync` method.

Queueing emails tolerates errors since the background job system has re-try mechanism to overcome temporary network/server problems.

See the [background jobs document](Background-Jobs.md) for more about the background job system.

## Email Settings

Email sending uses the [setting system](Settings.md) to define settings and get the values of these settings on the runtime. `Volo.Abp.Emailing.EmailSettingNames` defines constants for the setting names, just listed below:

\* **\*\*Abp.Mailing.DefaultFromAddress\*\***: Used as the sender's email address when you don't specify a sender when sending emails (just like in the example above).

\* **\*\*Abp.Mailing.DefaultFromDisplayName\*\***: Used as the sender's display name when you don't specify a sender when sending emails (just like in the example above).

\* **\*\*Abp.Mailing.Smtp.Host\*\***: The IP/Domain of the SMTP server (default: 127.0.0.1).

\* **\*\*Abp.Mailing.Smtp.Port\*\***: The Port of the SMTP server (default: 25).

\* **\*\*Abp.Mailing.Smtp.UserName\*\***: Username, if the SMTP server requires authentication.

\* **\*\*Abp.Mailing.Smtp.Password\*\***: Password, if the SMTP server requires authentication. \*\*This value is encrypted \*\*(see the section below).

\* **\*\*Abp.Mailing.Smtp.Domain\*\***: Domain for the username, if the SMTP server requires authentication.

\* **\*\*Abp.Mailing.Smtp.EnableSsl\*\***: A value that indicates if the SMTP server uses SSL or not ("true" or "false". Default: "false").

\* **\*\*Abp.Mailing.Smtp.UseDefaultCredentials\*\***: If true, uses default credentials instead of the provided username and password ("true" or "false". Default: "true").

Email settings can be managed from the *\*Settings Page\** of the [Setting Management](Modules/Setting-Management.md) module:

![email-settings](images/email-settings.png)

> Setting Management module is already installed if you've created your solution from the ABP Startup template.

If you don't use the Setting Management module, you can simply define the settings inside your `appsettings.json` file:

````json

"Settings": {

"Abp.Mailing.Smtp.Host": "127.0.0.1",

"Abp.Mailing.Smtp.Port": "25",

"Abp.Mailing.Smtp.UserName": "",

"Abp.Mailing.Smtp.Password": "",

"Abp.Mailing.Smtp.Domain": "",

"Abp.Mailing.Smtp.EnableSsl": "false",

"Abp.Mailing.Smtp.UseDefaultCredentials": "true",

"Abp.Mailing.DefaultFromAddress": "noreply@abp.io",

"Abp.Mailing.DefaultFromDisplayName": "ABP application"

}

````

You can set/change these settings programmatically using the `ISettingManager` and store values in a database. See the [setting system document](Settings.md) to understand the setting system better.

### Encrypt the SMTP Password

*\*Abp.Mailing.Smtp.Password\** must be an **\*\*encrypted\*\*** value. If you use the `ISettingManager` to set the password, you don't have to worry. It internally encrypts the values on set and decrypts on get.

If you use the `appsettings.json` to store the password, you should manually inject the `ISettingEncryptionService` and use its `Encrypt` method to obtain an encrypted value. This can be done by creating a simple code in your application. Then you can delete the code. As better, you can create a UI in your application to configure the email settings. In this case, you can directly use the `ISettingManager` without worrying the encryption.

### ISmtpEmailSenderConfiguration

If you don't want to use the setting system to store the email sending configuration, you can replace the `ISmtpEmailSenderConfiguration` service with your own implementation to get the configuration from any other source. `ISmtpEmailSenderConfiguration` is implemented by the `SmtpEmailSenderConfiguration` by default, which gets the configuration from the setting system as explained above.

## Text Template Integration

ABP Framework provides a strong and flexible [text templating system](Text-Templating.md). You can use the text templating system to create dynamic email contents. Inject the `ITemplateRenderer` and use the `RenderAsync` to render a template. Then use the result as the email body.

While you can define and use your own text templates, email sending system provides two simple built-in text templates.

**\*\*Example: Use the standard and simple message template to send emails\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

using Volo.Abp.Emailing.Templates;

using Volo.Abp.TextTemplating;

namespace Acme.BookStore.Web

{

public class MyService : ITransientDependency

{

private readonly IEmailSender \_emailSender;

private readonly ITemplateRenderer \_templateRenderer;

public MyService(

IEmailSender emailSender,

ITemplateRenderer templateRenderer)

{

\_emailSender = emailSender;

\_templateRenderer = templateRenderer;

}

public async Task DoItAsync()

{

var body = await \_templateRenderer.RenderAsync(

StandardEmailTemplates.Message,

new

{

message = "This is email body..."

}

);

await \_emailSender.SendAsync(

"target-address@domain.com",

"Email subject",

body

);

}

}

}

````

The resulting email body will be shown below:

````html

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

This is email body...

</body>

</html>

````

Emailing system defines the built-in text templates with the given names:

"**\*\*Abp.StandardEmailTemplates.Message\*\***" is simplest template that has a text message:

````html

{%{{{model.message}}}%}

````

This template uses the "Abp.StandardEmailTemplates.Layout" as its layout.

"**\*\*Abp.StandardEmailTemplates.Layout\*\***" is a simple template to provide an HTML document layout:

````html

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

{%{{{content}}}%}

</body>

</html>

````

The final rendered message was shown above.

> These template names are contants defined in the `Volo.Abp.Emailing.Templates.StandardEmailTemplates` class.

### Overriding/Replacing the Standard Templates

You typically want to replace the standard templates with your own ones, so you can prepare a branded email messages. To do that, you can use the power of the [virtual file system](Virtual-File-System.md) (VFS) or replace them in your own template definition provider.

Pathes of the templates in the virtual file system are shown below:

\* `/Volo/Abp/Emailing/Templates/Layout.tpl`

\* `/Volo/Abp/Emailing/Templates/Message.tpl`

If you add files to the same location in the virtual file system, your files will override them.

Templates are inline localized, that means you can take the power of the [localization system](Localization.md) to make your templates multi-cultural.

See the [text templating system](Text-Templating.md) document for details.

> Notice that you can define and use your own templates for your application, rather than using the standard simple templates. These standard templates are mostly for reusable modules where they don't define their own templates but rely on the built-in ones. This makes easy to customize emails sent by the used modules, by just overriding the standard email layout template.

## NullEmailSender

`NullEmailSender` is a built-in class that implements the `IEmailSender`, but writes email contents to the [standard log system](Logging.md), rathen than actually sending the emails.

This class can be useful especially in development time where you generally don't want to send real emails. The [application startup template](Startup-Templates/Application.md) already uses this class in the **\*\*DEBUG mode\*\*** with the following configuration in the domain layer:

````csharp

#if DEBUG

context.Services.Replace(ServiceDescriptor.Singleton<IEmailSender, NullEmailSender>());

#endif

````

So, don't confuse if you don't receive emails on DEBUG mode. Emails will be sent as expected on production (RELEASE mode). Remove these lines if you want to send real emails on DEBUG too.

## See Also

\* [MailKit integration for sending emails](MailKit.md)

### MailKit Integration

# MailKit Integration

[MailKit](http://www.mimekit.net/) is a cross-platform, popular open source mail client library for .net. ABP Framework provides an integration package to use the MailKit as the [email sender](Emailing.md).

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package. Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.MailKit

````

If you haven't done it yet, you first need to install the ABP CLI. For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.MailKit).

## Sending Emails

### IEmailSender

[Inject](Dependency-Injection.md) the standard `IEmailSender` into any service and use the `SendAsync` method to send emails. See the [email sending document](Emailing.md) for details.

> `IEmailSender` is the suggested way to send emails even if you use MailKit, since it makes your code provider independent.

### IMailKitSmtpEmailSender

MailKit package also exposes the `IMailKitSmtpEmailSender` service that extends the `IEmailSender` by adding the `BuildClientAsync()` method. This method can be used to obtain a `MailKit.Net.Smtp.SmtpClient` object that can be used to perform MailKit specific operations.

## Configuration

MailKit integration package uses the same settings defined by the email sending system. So, refer to the [email sending document](Emailing.md) for the settings.

In addition to the standard settings, this package defines `AbpMailKitOptions` as a simple [options](Options.md) class. This class defines only one options:

\* **\*\*SecureSocketOption\*\***: Used to set one of the `SecureSocketOptions`. Default: `null` (uses the defaults).

**\*\*Example: Use *\*SecureSocketOptions.SslOnConnect\**\*\***

````csharp

Configure<AbpMailKitOptions>(options =>

{

options.SecureSocketOption = SecureSocketOptions.SslOnConnect;

});

````

Refer to the [MailKit documentation](http://www.mimekit.net/) to learn more about this option.

## See Also

\* [Email sending](Emailing.md)

## Event Bus

### Overall

# Event Bus

An event bus is a mediator that transfers a message from a sender to a receiver. In this way, it provides a loosely coupled communication way between objects, services and applications.

## Event Bus Types

ABP Framework provides two type of event buses;

\* **\*\*[**Local Event Bus**](Local-Event-Bus.md)\*\*** is suitable for in-process messaging.

\* **\*\*[**Distributed Event Bus**](Distributed-Event-Bus.md)\*\*** is suitable for inter-process messaging, like microservices publishing and subscribing to distributed events.

### Local Event Bus

# Local Event Bus

The Local Event Bus allows services to publish and subscribe to **\*\*in-process events\*\***. That means it is suitable if two services (publisher and subscriber) are running in the same process.

## Publishing Events

There are two ways of publishing local events explained in the following sections.

### Publishing Events Using the ILocalEventBus

`ILocalEventBus` can be [injected](Dependency-Injection.md) and used to publish a local event.

**\*\*Example: Publish a local event when the stock count of a product changes\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus.Local;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly ILocalEventBus \_localEventBus;

public MyService(ILocalEventBus localEventBus)

{

\_localEventBus = localEventBus;

}

public virtual async Task ChangeStockCountAsync(Guid productId, int newCount)

{

//TODO: IMPLEMENT YOUR LOGIC...

//PUBLISH THE EVENT

await \_localEventBus.PublishAsync(

new StockCountChangedEvent

{

ProductId = productId,

NewCount = newCount

}

);

}

}

}

````

`PublishAsync` method gets a single parameter: the event object, which is responsible to hold the data related to the event. It is a simple plain class:

````csharp

using System;

namespace AbpDemo

{

public class StockCountChangedEvent

{

public Guid ProductId { get; set; }

public int NewCount { get; set; }

}

}

````

Even if you don't need to transfer any data, you need to create a class (which is an empty class in this case).

### Publishing Events Inside Entity / Aggregate Root Classes

[Entities](Entities.md) can not inject services via dependency injection, but it is very common to publish local events inside entity / aggregate root classes.

**\*\*Example: Publish a local event inside an aggregate root method\*\***

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace AbpDemo

{

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

public int StockCount { get; private set; }

private Product() { }

public Product(Guid id, string name)

: base(id)

{

Name = name;

}

public void ChangeStockCount(int newCount)

{

StockCount = newCount;

//ADD an EVENT TO BE PUBLISHED

AddLocalEvent(

new StockCountChangedEvent

{

ProductId = Id,

NewCount = newCount

}

);

}

}

}

````

`AggregateRoot` class defines the `AddLocalEvent` to add a new local event, that is published when the aggregate root object is saved (created, updated or deleted) into the database.

> Tip: If an entity publishes such an event, it is a good practice to change the related properties in a controlled manner, just like the example above - `StockCount` can only be changed by the `ChangeStockCount` method which guarantees publishing the event.

#### IGeneratesDomainEvents Interface

Actually, adding local events are not unique to the `AggregateRoot` class. You can implement `IGeneratesDomainEvents` for any entity class. But, `AggregateRoot` implements it by default and makes it easy for you.

> It is not suggested to implement this interface for entities those are not aggregate roots, since it may not work for some database providers for such entities. It works for EF Core, but not works for MongoDB for example.

#### How It Was Implemented?

Calling the `AddLocalEvent` doesn't immediately publish the event. The event is published when you save changes to the database;

\* For EF Core, it is published on `DbContext.SaveChanges`.

\* For MongoDB, it is published when you call repository's `InsertAsync`, `UpdateAsync` or `DeleteAsync` methods (since MongoDB has not a change tracking system).

## Subscribing to Events

A service can implement the `ILocalEventHandler<TEvent>` to handle the event.

**\*\*Example: Handle the** `StockCountChangedEvent` **defined above\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus;

namespace AbpDemo

{

public class MyHandler

: ILocalEventHandler<StockCountChangedEvent>,

ITransientDependency

{

public async Task HandleEventAsync(StockCountChangedEvent eventData)

{

//TODO: your code that does something on the event

}

}

}

````

That's all. `MyHandler` is **\*\*automatically discovered\*\*** by the ABP Framework and `HandleEventAsync` is called whenever a `StockCountChangedEvent` occurs. You can inject any service and perform any required logic in your handler class.

\* **\*\*One or more handlers\*\*** can subscribe to the same event.

\* A single event handler class can **\*\*subscribe to multiple events\*\*** by implementing the `ILocalEventHandler<TEvent>` interface for each event type.

If you perform **\*\*database operations\*\*** and use the [repositories](Repositories.md) inside the event handler, you may need to create a [unit of work](Unit-Of-Work.md), because some repository methods need to work inside an **\*\*active unit of work\*\***. Make the handle method `virtual` and add a `[UnitOfWork]` attribute for the method, or manually use the `IUnitOfWorkManager` to create a unit of work scope.

> The handler class must be registered to the dependency injection (DI). The sample above uses the `ITransientDependency` to accomplish it. See the [DI document](Dependency-Injection.md) for more options.

### LocalEventHandlerOrder Attribute

`LocalEventHandlerOrder` attribute can be used to set the execution order for the event handlers, which can be helpful if you want to handle your event handlers in a specific order.

````csharp

[LocalEventHandlerOrder(-1)]

public class MyHandler

: ILocalEventHandler<StockCountChangedEvent>,

ITransientDependency

{

public async Task HandleEventAsync(StockCountChangedEvent eventData)

{

//TODO: your code that does something on the event

}

}

````

> By default, all event handlers have an order value of 0. Thus, if you want to take certain event handlers to be executed before other event handlers, you can set the order value as a negative value.

#### LocalEventHandlerOrderAttribute Properties

\* `Order` (`int`): Used to set the execution order for a certain event handler.

### Transaction & Exception Behavior

Event handlers are always executed in the same [unit of work](Unit-Of-Work.md) scope, that means in the same database transaction with the code that published the event. If an event handler throws an exception, the unit of work (database transaction) is rolled back. So, **\*\*use try-catch yourself\*\*** in the event handler if you want to hide the error.

When you call `ILocalEventBus.PublishAsync`, the event handlers are not immediately executed. Instead, they are executed just before the current unit of work completed (an unhandled exception in the handler still rollbacks the current unit of work). If you want to immediately execute the handlers, set the optional `onUnitOfWorkComplete` parameter to `false`.

> Keeping the default behavior is recommended unless you don't have a unique requirement. `onUnitOfWorkComplete` option is not available when you publish events inside entity / aggregate root classes (see the *\*Publishing Events Inside Entity / Aggregate Root Classes\** section).

## Pre-Built Events

It is very common to **\*\*publish events on entity create, update and delete\*\*** operations. ABP Framework **\*\*automatically\*\*** publish these events for all entities. You can just subscribe to the related event.

**\*\*Example: Subscribe to an event that published when a user was created\*\***

````csharp

using System.Threading.Tasks;

using Microsoft.AspNetCore.Identity;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Entities.Events;

using Volo.Abp.EventBus;

namespace AbpDemo

{

public class MyHandler

: ILocalEventHandler<EntityCreatedEventData<IdentityUser>>,

ITransientDependency

{

public async Task HandleEventAsync(

EntityCreatedEventData<IdentityUser> eventData)

{

var userName = eventData.Entity.UserName;

var email = eventData.Entity.Email;

//...

}

}

}

````

This class subscribes to the `EntityCreatedEventData<IdentityUser>`, which is published just after a user was created (but before the current transaction is completed). For example, you may want to send a "Welcome" email to the new user.

The pre-built event types are;

\* `EntityCreatedEventData<T>` is published just after an entity was successfully created.

\* `EntityUpdatedEventData<T>` is published just after an entity was successfully updated.

\* `EntityDeletedEventData<T>` is published just after an entity was successfully deleted.

\* `EntityChangedEventData<T>` is published just after an entity was successfully created, updated or deleted. It can be a shortcut if you need to listen any type of change - instead of subscribing to the individual events.

### How It Was Implemented?

Pre-build events are published when you save changes to the database;

\* For EF Core, they are published on `DbContext.SaveChanges`.

\* For MongoDB, they are published when you call repository's `InsertAsync`, `UpdateAsync` or `DeleteAsync` methods (since MongoDB has not a change tracking system).

## See Also

\* [Distributed Event Bus](Distributed-Event-Bus.md)

### Distributed Event Bus

# Distributed Event Bus

Distributed Event bus system allows to **\*\*publish\*\*** and **\*\*subscribe\*\*** to events that can be **\*\*transferred across application/service boundaries\*\***. You can use the distributed event bus to asynchronously send and receive messages between **\*\*microservices\*\*** or **\*\*applications\*\***.

## Providers

Distributed event bus system provides an **\*\*abstraction\*\*** that can be implemented by any vendor/provider. There are four providers implemented out of the box:

\* `LocalDistributedEventBus` is the default implementation that implements the distributed event bus to work as in-process. Yes! The **\*\*default implementation works just like the [**local event bus**](Local-Event-Bus.md)\*\***, if you don't configure a real distributed provider.

\* `AzureDistributedEventBus` implements the distributed event bus with the [Azure Service Bus](https://azure.microsoft.com/en-us/services/service-bus/). See the [Azure Service Bus integration document](Distributed-Event-Bus-Azure-Integration.md) to learn how to configure it.

\* `RabbitMqDistributedEventBus` implements the distributed event bus with the [RabbitMQ](https://www.rabbitmq.com/). See the [RabbitMQ integration document](Distributed-Event-Bus-RabbitMQ-Integration.md) to learn how to configure it.

\* `KafkaDistributedEventBus` implements the distributed event bus with the [Kafka](https://kafka.apache.org/). See the [Kafka integration document](Distributed-Event-Bus-Kafka-Integration.md) to learn how to configure it.

\* `RebusDistributedEventBus` implements the distributed event bus with the [Rebus](http://mookid.dk/category/rebus/). See the [Rebus integration document](Distributed-Event-Bus-Rebus-Integration.md) to learn how to configure it.

Using a local event bus as default has a few important advantages. The most important one is that: It allows you to write your code compatible to distributed architecture. You can write a monolithic application now that can be split into microservices later. It is a good practice to communicate between bounded contexts (or between application modules) via distributed events instead of local events.

For example, [pre-built application modules](Modules/Index.md) is designed to work as a service in a distributed system while they can also work as a module in a monolithic application without depending an external message broker.

## Publishing Events

There are two ways of publishing distributed events explained in the following sections.

### Using IDistributedEventBus to Publish Events

`IDistributedEventBus` can be [injected](Dependency-Injection.md) and used to publish a distributed event.

**\*\*Example: Publish a distributed event when the stock count of a product changes\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus.Distributed;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IDistributedEventBus \_distributedEventBus;

public MyService(IDistributedEventBus distributedEventBus)

{

\_distributedEventBus = distributedEventBus;

}

public virtual async Task ChangeStockCountAsync(Guid productId, int newCount)

{

await \_distributedEventBus.PublishAsync(

new StockCountChangedEto

{

ProductId = productId,

NewCount = newCount

}

);

}

}

}

````

`PublishAsync` method gets the event object, which is responsible to hold the data related to the event. It is a simple plain class:

````csharp

using System;

namespace AbpDemo

{

[EventName("MyApp.Product.StockChange")]

public class StockCountChangedEto

{

public Guid ProductId { get; set; }

public int NewCount { get; set; }

}

}

````

Even if you don't need to transfer any data, you need to create a class (which is an empty class in this case).

> `Eto` is a suffix for **\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bjects we use by convention. While it is not required, we find it useful to identify such event classes (just like [DTOs](Data-Transfer-Objects.md) on the application layer).

#### Event Name

`EventName` attribute is optional, but suggested. If you don't declare it for an event type (ETO class), the event name will be the full name of the event class, `AbpDemo.StockCountChangedEto` in this case.

#### About Serialization for the Event Objects

Event transfer objects (ETOs) **\*\*must be serializable\*\*** since they will be serialized/deserialized to JSON or other format when it is transferred to out of the process.

Avoid circular references, polymorphism, private setters and provide default (empty) constructors if you have any other constructor as a good practice (while some serializers may tolerate it), just like the DTOs.

### Publishing Events Inside Entity / Aggregate Root Classes

[Entities](Entities.md) can not inject services via dependency injection, but it is very common to publish distributed events inside entity / aggregate root classes.

**\*\*Example: Publish a distributed event inside an aggregate root method\*\***

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace AbpDemo

{

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

public int StockCount { get; private set; }

private Product() { }

public Product(Guid id, string name)

: base(id)

{

Name = name;

}

public void ChangeStockCount(int newCount)

{

StockCount = newCount;

//ADD an EVENT TO BE PUBLISHED

AddDistributedEvent(

new StockCountChangedEto

{

ProductId = Id,

NewCount = newCount

}

);

}

}

}

````

`AggregateRoot` class defines the `AddDistributedEvent` to add a new distributed event, that is published when the aggregate root object is saved (created, updated or deleted) into the database.

> If an entity publishes such an event, it is a good practice to change the related properties in a controlled manner, just like the example above - `StockCount` can only be changed by the `ChangeStockCount` method which guarantees publishing the event.

#### IGeneratesDomainEvents Interface

Actually, adding distributed events are not unique to the `AggregateRoot` class. You can implement `IGeneratesDomainEvents` for any entity class. But, `AggregateRoot` implements it by default and makes it easy for you.

> It is not suggested to implement this interface for entities those are not aggregate roots, since it may not work for some database providers for such entities. It works for EF Core, but not works for MongoDB for example.

#### How It Was Implemented?

Calling the `AddDistributedEvent` doesn't immediately publish the event. The event is published when you save changes to the database;

\* For EF Core, it is published on `DbContext.SaveChanges`.

\* For MongoDB, it is published when you call repository's `InsertAsync`, `UpdateAsync` or `DeleteAsync` methods (since MongoDB has not a change tracking system).

## Subscribing to Events

A service can implement the `IDistributedEventHandler<TEvent>` to handle the event.

**\*\*Example: Handle the** `StockCountChangedEto` **defined above\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus.Distributed;

namespace AbpDemo

{

public class MyHandler

: IDistributedEventHandler<StockCountChangedEto>,

ITransientDependency

{

public async Task HandleEventAsync(StockCountChangedEto eventData)

{

var productId = eventData.ProductId;

}

}

}

````

That's all.

\* `MyHandler` is **\*\*automatically discovered\*\*** by the ABP Framework and `HandleEventAsync` is called whenever a `StockCountChangedEto` event occurs.

\* If you are using a distributed message broker, like RabbitMQ, ABP automatically **\*\*subscribes to the event on the message broker\*\***, gets the message, executes the handler.

\* It sends **\*\*confirmation (ACK)\*\*** to the message broker if the event handler was successfully executed (did not throw any exception).

You can inject any service and perform any required logic here. A single event handler class can **\*\*subscribe to multiple events\*\*** but implementing the `IDistributedEventHandler<TEvent>` interface for each event type.

If you perform **\*\*database operations\*\*** and use the [repositories](Repositories.md) inside the event handler, you may need to create a [unit of work](Unit-Of-Work.md), because some repository methods need to work inside an **\*\*active unit of work\*\***. Make the handle method `virtual` and add a `[UnitOfWork]` attribute for the method, or manually use the `IUnitOfWorkManager` to create a unit of work scope.

> The handler class must be registered to the dependency injection (DI). The sample above uses the `ITransientDependency` to accomplish it. See the [DI document](Dependency-Injection.md) for more options.

## Monitoring Distributed Events

The ABP Framework allows you to stay informed when your application **\*\*receives\*\*** or **\*\*sends\*\*** a distributed event. This capability enables you to track the event flow within your application and take appropriate actions based on the received or sent distributed events.

### Received Events

The `DistributedEventReceived` local event is published when your application receives an event from the distributed event bus. `DistributedEventReceived` class has the following fields:

- **\*\***`Source`**:\*\*** It represents the source of the distributed event. Source can be `Direct`, `Inbox`, `Outbox`.

- **\*\***`EventName`**:\*\*** It represents the [name](#event-name) of the event received.

- **\*\***`EventData`**:\*\*** It represents the actual data associated with the event received. Since it is of type `object`, it can hold any type of data.

**\*\*Example: Get informed when your application receives an event from the distributed event bus\*\***

```csharp

public class DistributedEventReceivedHandler : ILocalEventHandler<DistributedEventReceived>, ITransientDependency

{

public async Task HandleEventAsync(DistributedEventReceived eventData)

{

// TODO: IMPLEMENT YOUR LOGIC...

}

}

```

### Sent Events

The `DistributedEventSent` local event is published when your application sends an event to the distributed event bus. `DistributedEventSent` class has the following fields:

- **\*\***`Source`**:\*\*** It represents the source of the distributed event. Source can be `Direct`, `Inbox`, `Outbox`.

- **\*\***`EventName`**:\*\*** It represents the [name](#event-name) of the event sent.

- **\*\***`EventData`**:\*\*** It represents the actual data associated with the event sent. Since it is of type `object`, it can hold any type of data.

**\*\*Example: Get informed when your application sends an event to the distributed event bus\*\***

```csharp

public class DistributedEventSentHandler : ILocalEventHandler<DistributedEventSent>, ITransientDependency

{

public async Task HandleEventAsync(DistributedEventSent eventData)

{

// TODO: IMPLEMENT YOUR LOGIC...

}

}

```

You can seamlessly integrate event-tracking capabilities into your application by subscribing to the `DistributedEventReceived` and `DistributedEventSent` local events as above examples. This empowers you to effectively monitor the messaging flow, diagnose any potential issues, and gain valuable insights into the behavior of your distributed messaging system.

## Pre-Defined Events

ABP Framework **\*\*automatically publishes\*\*** distributed events for **\*\*create, update and delete\*\*** operations for an [entity](Entities.md) once you configure it.

### Event Types

There are three pre-defined event types:

\* `EntityCreatedEto<T>` is published when an entity of type `T` was created.

\* `EntityUpdatedEto<T>` is published when an entity of type `T` was updated.

\* `EntityDeletedEto<T>` is published when an entity of type `T` was deleted.

These types are generics. `T` is actually the type of the **\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bject (ETO) rather than the type of the entity. Because, an entity object can not be transferred as a part of the event data. So, it is typical to define a ETO class for an entity class, like `ProductEto` for `Product` entity.

### Subscribing to the Events

Subscribing to the auto events is same as subscribing a regular distributed event.

**\*\*Example: Get notified once a product updated\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Entities.Events.Distributed;

using Volo.Abp.EventBus.Distributed;

namespace AbpDemo

{

public class MyHandler :

IDistributedEventHandler<EntityUpdatedEto<ProductEto>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityUpdatedEto<ProductEto> eventData)

{

var productId = eventData.Entity.Id;

//TODO

}

}

}

````

\* `MyHandler` implements the `IDistributedEventHandler<EntityUpdatedEto<ProductEto>>`.

\* It is required to register your handler class to the [dependency injection](Dependency-Injection.md) system. Implementing `ITransientDependency` like in this example is an easy way.

### Configuration

You can configure the `AbpDistributedEntityEventOptions` in the `ConfigureServices` of your [module](Module-Development-Basics.md) to add a selector.

**\*\*Example: Configuration samples\*\***

````csharp

Configure<AbpDistributedEntityEventOptions>(options =>

{

//Enable for all entities

options.AutoEventSelectors.AddAll();

//Enable for a single entity

options.AutoEventSelectors.Add<Product>();

//Enable for all entities in a namespace (and child namespaces)

options.AutoEventSelectors.AddNamespace("MyProject.Products");

//Custom predicate expression that should return true to select a type

options.AutoEventSelectors.Add(

type => type.Namespace.StartsWith("MyProject.")

);

});

````

\* The last one provides flexibility to decide if the events should be published for the given entity type. Returns `true` to accept a `Type`.

You can add more than one selector. If one of the selectors match for an entity type, then it is selected.

### Event Transfer Object

Once you enable **\*\*auto events\*\*** for an entity, ABP Framework starts to publish events on the changes on this entity. If you don't specify a corresponding **\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bject (ETO) for the entity, ABP Framework uses a standard type, named `EntityEto`, which has only two properties:

\* `EntityType` (`string`): Full name (including namespace) of the entity class.

\* `KeysAsString` (`string`): Primary key(s) of the changed entity. If it has a single key, this property will be the primary key value. For a composite key, it will contain all keys separated by `,` (comma).

So, you can implement the `IDistributedEventHandler<EntityUpdatedEto<EntityEto>>` to subscribe the update events. However, it is not a good approach to subscribe to such a generic event, because you handle the update events for all entities in a single handler (since they all use the same ETO object). You can define the corresponding ETO type for the entity type.

**\*\*Example: Declare to use** `ProductEto` **for the** `Product` **entity\*\***

````csharp

public class ProductEto

{

public Guid Id { get; set; }

public string Name { get; set; }

public float Price { get; set; }

}

````

Then you can use the `AbpDistributedEntityEventOptions.EtoMappings` option to map your `Product` entity to the `ProductEto`:

````csharp

Configure<AbpDistributedEntityEventOptions>(options =>

{

options.AutoEventSelectors.Add<Product>();

options.EtoMappings.Add<Product, ProductEto>();

});

````

This example;

\* Adds a selector to allow to publish the create, update and delete events for the `Product` entity.

\* Configure to use the `ProductEto` as the event transfer object to publish for the `Product` related events.

> Distributed event system use the [object to object mapping](Object-To-Object-Mapping.md) system to map `Product` objects to `ProductEto` objects. So, you need to configure the object mapping (`Product` -> `ProductEto`) too. You can check the [object to object mapping document](Object-To-Object-Mapping.md) to learn how to do it.

## Entity Synchronizer

In a distributed (or microservice) system, it is typical to subscribe to change events for an [entity](Entities.md) type of another service, so you can get notifications when the subscribed entity changes. In that case, you can use ABP's Pre-Defined Events as explained in the previous section.

If your purpose is to store your local copies of a remote entity, you typically subscribe to create, update and delete events of the remote entity and update your local database in your event handler. ABP provides a pre-built `EntitySynchronizer` base class to make that operation easier for you.

Assume that there is a `Product` entity (probably an aggregate root entity) in a Catalog microservice, and you want to keep copies of the products in your Ordering microservice, with a local `OrderProduct` entity. In practice, properties of the `OrderProduct` class will be a subset of the `Product` properties, because not all the product data is needed in the Ordering microservice (however, you can make a full copy if you need). Also, the `OrderProduct` entity may have additional properties that are populated and used in the Ordering microservice.

The first step to establish the synchronization is to define an ETO (Event Transfer Object) class in the Catalog microservice that is used to transfer the event data. Assuming the `Product` entity has a `Guid` key, your ETO can be as shown below:

````csharp

[EventName("product")]

public class ProductEto : EntityEto<Guid>

{

// Your Product properties here...

}

````

`ProductEto` can be put in a shared project (DLL) that is referenced by the Catalog and the Ordering microservices. Alternatively, you can put a copy of the `ProductEto` class in the Ordering microservice if you don't want to introduce a common project dependency between the services. In this case, the `EventName` attribute becomes critical to map the `ProductEto` classes across two services (you should use the same event name).

Once you define an ETO class, you should configure the ABP Framework to publish auto (create, update and delete) events for the `Product` entity, as explained in the previous section:

````csharp

Configure<AbpDistributedEntityEventOptions>(options =>

{

options.AutoEventSelectors.Add<Product>();

options.EtoMappings.Add<Product, ProductEto>();

});

````

Finally, you should create a class in the Ordering microservice, that is derived from the `EntitySynchronizer` class:

````csharp

public class ProductSynchronizer : EntitySynchronizer<OrderProduct, Guid, ProductEto>

{

public ProductSynchronizer(

IObjectMapper objectMapper,

IRepository<OrderProduct, Guid> repository

) : base(objectMapper, repository)

{

}

}

````

The main point of this class is it subscribes to the create, update and delete events of the source entity and updates the local entity in the database. It uses the [Object Mapper](Object-To-Object-Mapping.md) system to create or update the `OrderProduct` objects from the `ProductEto` objects. So, you should also configure the object mapper to make it properly work. Otherwise, you should manually perform the object mapping by overriding the `MapToEntityAsync(TSourceEntityEto)` and `MapToEntityAsync(TSourceEntityEto,TEntity)` methods in your `ProductSynchronizer` class.

If your entity has a composite primary key (see the [Entities document](Entities.md)), then you should inherit from the `EntitySynchronizer<TEntity, TSourceEntityEto>` class (just don't use the `Guid` generic argument in the previous example) and implement `FindLocalEntityAsync` to find the entity in your local database using the `Repository`.

`EntitySynchronizer` is compatible with the *\*Entity Versioning\** system (see the [Entities document](Entities.md)). So, it works as expected even if the events are received as disordered. If the entity's version in your local database is newer than the entity in the received event, then the event is ignored. You should implement the `IHasEntityVersion` interface for the entity and ETO classes (for this example, you should implement for the `Product`, `ProductEto` and `OrderProduct` classes).

If you want to ignore some type of change events, you can set `IgnoreEntityCreatedEvent`, `IgnoreEntityUpdatedEvent` and `IgnoreEntityDeletedEvent` in the constructor of your class. Example:

````csharp

public class ProductSynchronizer

: EntitySynchronizer<OrderProduct, Guid, ProductEto>

{

public ProductSynchronizer(

IObjectMapper objectMapper,

IRepository<OrderProduct, Guid> repository

) : base(objectMapper, repository)

{

IgnoreEntityDeletedEvent = true;

}

}

````

> Notice that the `EntitySynchronizer` can only create/update the entities after you use it. If you have an existing system with existing data, you should manually copy the data for one time, because the `EntitySynchronizer` starts to work.

## Transaction and Exception Handling

Distributed event bus works in-process (since default implementation is `LocalDistributedEventBus`) unless you configure an actual provider (e.g. [Kafka](Distributed-Event-Bus-Kafka-Integration.md) or [RabbitMQ](Distributed-Event-Bus-RabbitMQ-Integration.md)). In-process event bus always executes event handlers in the same [unit of work](Unit-Of-Work.md) scope that you publishes the events in. That means, if an event handler throws an exception, then the related unit of work (the database transaction) is rolled back. In this way, your application logic and event handling logic becomes transactional (atomic) and consistent. If you want to ignore errors in an event handler, you must use a `try-catch` block in your handler and shouldn't re-throw the exception.

When you switch to an actual distributed event bus provider (e.g. [Kafka](Distributed-Event-Bus-Kafka-Integration.md) or [RabbitMQ](Distributed-Event-Bus-RabbitMQ-Integration.md)), then the event handlers will be executed in different processes/applications as their purpose is to create distributed systems. In this case, the only way to implement transactional event publishing is to use the outbox/inbox patterns as explained in the *\*Outbox / Inbox for Transactional Events\** section.

If you don't configure outbox/inbox pattern or use the `LocalDistributedEventBus`, then events are published at the end of the unit of work by default, just before the unit of work is completed (that means throwing exception in an event handler still rollbacks the unit of work), even if you publish them in the middle of unit of work. If you want to immediately publish the event, you can set `onUnitOfWorkComplete` to `false` while using `IDistributedEventBus.PublishAsync` method.

> Keeping the default behavior is recommended unless you don't have a unique requirement. `onUnitOfWorkComplete` option is not available when you publish events inside entity / aggregate root classes (see the *\*Publishing Events Inside Entity / Aggregate Root Classes\** section).

## Outbox / Inbox for Transactional Events

The **\*\*[**transactional outbox pattern**](https://microservices.io/patterns/data/transactional-outbox.html)\*\*** is used to publishing distributed events within the **\*\*same transaction\*\*** that manipulates the application's database. When you enable outbox, distributed events are saved into the database inside the same transaction with your data changes, then sent to the actual message broker by a separate [background worker](Background-Workers.md) with a re-try system. In this way, it ensures the consistency between your database state and the published events.

The **\*\*transactional inbox pattern\*\***, on the other hand, saves incoming events into database first. Then (in a [background worker](Background-Workers.md)) executes the event handler in a transactional manner and removes the event from the inbox queue in the same transaction. It ensures that the event is only executed one time by keeping the processed messages for a while and discarding the duplicate events received from the message broker.

Enabling the event outbox and inbox systems require a few manual steps for your application. Please apply the instructions in the following sections to make them running.

> Outbox and Inbox can be separately enabled and configured, so you may only use one of them if you want.

### Pre-requirements

\* The outbox/inbox system uses the distributed lock system to handle concurrency when you run multiple instances of your application/service. So, you should **\*\*configure the distributed lock system\*\*** with one of the providers as [explained in this document](Distributed-Locking.md).

\* The outbox/inbox system supports [Entity Framework Core](Entity-Framework-Core.md) (EF Core) and [MongoDB](MongoDB.md) **\*\*database providers\*\*** out of the box. So, your applications should use one of these database providers. For other database providers, see the *\*Implementing a Custom Database Provider\** section.

> If you are using MongoDB, be sure that you enabled multi-document database transactions that was introduced in MongoDB version 4.0. See the *\*Transactions\** section of the [MongoDB](MongoDB.md) document.

### Enabling event outbox

Open your `DbContext` class (EF Core or MongoDB), implement the `IHasEventOutbox` interface. You should end up by adding a `DbSet` property into your `DbContext` class:

```csharp

public DbSet<OutgoingEventRecord> OutgoingEvents { get; set; }

```

Add the following lines inside the `OnModelCreating` method of your `DbContext` class (only for EF Core):

```csharp

builder.ConfigureEventOutbox();

```

For EF Core, use the standard `Add-Migration` and `Update-Database` commands to apply changes into your database (you can skip this step for MongoDB). If you want to use the command-line terminal, run the following commands in the root directory of the database integration project:

```bash

dotnet ef migrations add "Added\_Event\_Outbox"

dotnet ef database update

```

Finally, write the following configuration code inside the `ConfigureServices` method of your [module class](Module-Development-Basics.md) (replace `YourDbContext` with your own `DbContext` class):

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Outboxes.Configure(config =>

{

config.UseDbContext<YourDbContext>();

});

});

````

> **\*\*IMPORTANT\*\***: Outbox sending service uses distributed locks to ensure only a single instance of your application consumes the outbox queue concurrently. Distributed locking key should be unique per database. The `config` object (in the preceding code example) has a `DatabaseName` property, which is used in the distributed lock key to ensure the uniqueness. `DatabaseName` is automatically set by the `UseDbContext` method, getting the database name from the `ConnectionStringName` attribute of the `YourDbContext` class. So, if you have multiple databases in your system, ensure that you use the same connection string name for the same database, but different connection string names for different databases. If you can't ensure that, you can manually set `config.DatabaseName` (after the `UseDbContext` line) to ensure that uniqueness.

### Enabling event inbox

Open your `DbContext` class (EF Core or MongoDB), implement the `IHasEventInbox` interface. You should end up by adding a `DbSet` property into your `DbContext` class:

```csharp

public DbSet<IncomingEventRecord> IncomingEvents { get; set; }

```

Add the following lines inside the `OnModelCreating` method of your `DbContext` class (only for EF Core):

```csharp

builder.ConfigureEventInbox();

```

For EF Core, use the standard `Add-Migration` and `Update-Database` commands to apply changes into your database (you can skip this step for MongoDB). If you want to use the command-line terminal, run the following commands in the root directory of the database integration project:

```bash

dotnet ef migrations add "Added\_Event\_Inbox"

dotnet ef database update

```

Finally, write the following configuration code inside the `ConfigureServices` method of your [module class](Module-Development-Basics.md) (replace `YourDbContext` with your own `DbContext` class):

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Inboxes.Configure(config =>

{

config.UseDbContext<YourDbContext>();

});

});

````

**\*\*IMPORTANT\*\***: Inbox processing service uses distributed locks to ensure only a single instance of your application consumes the inbox queue concurrently. Distributed locking key should be unique per database. The `config` object (in the preceding code example) has a `DatabaseName` property, which is used in the distributed lock key to ensure the uniqueness. `DatabaseName` is automatically set by the `UseDbContext` method, getting the database name from the `ConnectionStringName` attribute of the `YourDbContext` class. So, if you have multiple databases in your system, ensure that you use the same connection string name for the same database, but different connection string names for different databases. If you can't ensure that, you can manually set `config.DatabaseName` (after the `UseDbContext` line) to ensure that uniqueness.

### Additional Configuration

> The default configuration will be enough for most cases. However, there are some options you may want to set for outbox and inbox.

#### Outbox configuration

Remember how outboxes are configured:

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Outboxes.Configure(config =>

{

// TODO: Set options

});

});

````

Here, the following properties are available on the `config` object:

\* `IsSendingEnabled` (default: `true`): You can set to `false` to disable sending outbox events to the actual event bus. If you disable this, events can still be added to outbox, but not sent. This can be helpful if you have multiple applications (or application instances) writing to outbox, but use one of them to send the events.

\* `Selector`: A predicate to filter the event (ETO) types to be used for this configuration. Should return `true` to select the event. It selects all the events by default. This is especially useful if you want to ignore some ETO types from the outbox, or want to define named outbox configurations and group events within these configurations. See the *\*Named Configurations\** section.

\* `ImplementationType`: Type of the class that implements the database operations for the outbox. This is normally set when you call `UseDbContext` as shown before. See *\*Implementing a Custom Outbox/Inbox Database Provider\** section for advanced usages.

\* `DatabaseName`: Unique database name for the database that is used for this outbox configuration. See the **\*\*IMPORTANT\*\*** paragraph at the end of the *\*Enabling event outbox\** section.

#### Inbox configuration

Remember how inboxes are configured:

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Inboxes.Configure(config =>

{

// TODO: Set options

});

});

````

Here, the following properties are available on the `config` object:

\* `IsProcessingEnabled` (default: `true`): You can set to `false` to disable processing (handling) events in the inbox. If you disable this, events can still be received, but not executed. This can be helpful if you have multiple applications (or application instances), but use one of them to execute the event handlers.

\* `EventSelector`: A predicate to filter the event (ETO) types to be used for this configuration. This is especially useful if you want to ignore some ETO types from the inbox, or want to define named inbox configurations and group events within these configurations. See the *\*Named Configurations\** section.

\* `HandlerSelector`: A predicate to filter the event handled types (classes implementing the `IDistributedEventHandler<TEvent>` interface) to be used for this configuration. This is especially useful if you want to ignore some event handler types from inbox processing, or want to define named inbox configurations and group event handlers within these configurations. See the *\*Named Configurations\** section.

\* `ImplementationType`: Type of the class that implements the database operations for the inbox. This is normally set when you call `UseDbContext` as shown before. See *\*Implementing a Custom Outbox/Inbox Database Provider\** section for advanced usages.

\* `DatabaseName`: Unique database name for the database that is used for this outbox configuration. See the **\*\*IMPORTANT\*\*** paragraph at the end of the *\*Enabling event inbox\** section.

#### AbpEventBusBoxesOptions

`AbpEventBusBoxesOptions` can be used to fine-tune how inbox and outbox systems work. For most of the systems, using the defaults would be more than enough, but you can configure it to optimize your system when it is needed.

Just like all the [options classes](Options.md), `AbpEventBusBoxesOptions` can be configured in the `ConfigureServices` method of your [module class](Module-Development-Basics.md) as shown in the following code block:

````csharp

Configure<AbpEventBusBoxesOptions>(options =>

{

// TODO: configure the options

});

````

`AbpEventBusBoxesOptions` has the following properties to be configured:

\* `BatchPublishOutboxEvents`: Can be used to enable or disable batch publishing events to the message broker. Batch publishing works if it is supported by the distributed event bus provider. If not supported, events are sent one by one as the fallback logic. Keep it as enabled since it has a great performance gain wherever possible. Default value is `true` (enabled).

\* `PeriodTimeSpan`: The period of the inbox and outbox message processors to check if there is a new event in the database. Default value is 2 seconds (`TimeSpan.FromSeconds(2)`).

\* `CleanOldEventTimeIntervalSpan`: The event inbox system periodically checks and deletes the old processed events from the inbox in the database. You can set this value to determine the check period. Default value is 6 hours (`TimeSpan.FromHours(6)`).

\* `WaitTimeToDeleteProcessedInboxEvents`: Inbox events are not deleted from the database for a while even if they are successfully processed. This is for a system to prevent multiple process of the same event (if the event broker sends it twice). This configuration value determines the time to keep the processed events. Default value is 2 hours (`TimeSpan.FromHours(2)`).

\* `InboxWaitingEventMaxCount`: The maximum number of events to query at once from the inbox in the database. Default value is 1000.

\* `OutboxWaitingEventMaxCount`: The maximum number of events to query at once from the outbox in the database. Default value is 1000.

\* `DistributedLockWaitDuration`: ABP uses [distributed locking](Distributed-Locking.md) to prevent concurrent access to the inbox and outbox messages in the database, when running multiple instance of the same application. If an instance of the application can not obtain the lock, it tries after a duration. This is the configuration of that duration. Default value is 15 seconds (`TimeSpan.FromSeconds(15)`).

### Skipping Outbox

`IDistributedEventBus.PublishAsync` method provides an optional parameter, `useOutbox`, which is set to `true` by default. If you bypass outbox and immediately publish an event, you can set it to `false` for a specific event publishing operation.

### Advanced Topics

#### Named Configurations

> All the concepts explained in this section is also valid for inbox configurations. We will show examples only for outbox to keep the document shorter.

See the following outbox configuration code:

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Outboxes.Configure(config =>

{

//TODO

});

});

````

This is equivalent of the following code:

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Outboxes.Configure("Default", config =>

{

//TODO

});

});

````

`Default` is this code indicates the configuration name. If you don't specify it (like in the previous code block), `Default` is used as the configuration name.

That means you can define more than one configuration for outbox (also for inbox) with different names. ABP runs all the configured outboxes.

Multiple outboxes can be needed if your application have more than one database and you want to run different outbox queues for different databases. In this case, you can use the `Selector` option to decide the events should be handled by an outbox. See the *\*Additional Configurations\** section above.

#### Implementing a Custom Outbox/Inbox Database Provider

If your application or service is using a database provider other than [EF Core](Entity-Framework-Core.md) and [MongoDB](MongoDB.md), you should manually integrate outbox/inbox system with your database provider.

> Outbox and Inbox table/data must be stored in the same database with your application's data (since we want to create a single database transaction that includes application's database operations and outbox/inbox table operations). Otherwise, you should care about distributed (multi-database) transaction support which is not provided by most of the vendors and may require additional configuration.

ABP provides `IEventOutbox` and `IEventInbox` abstractions as extension point for the outbox/inbox system. You can create classes by implementing these interfaces and register them to [dependency injection](Dependency-Injection.md).

Once you implement your custom event boxes, you can configure `AbpDistributedEventBusOptions` to use your event box classes:

````csharp

Configure<AbpDistributedEventBusOptions>(options =>

{

options.Outboxes.Configure(config =>

{

config.ImplementationType = typeof(MyOutbox); //Your Outbox class

});

options.Inboxes.Configure(config =>

{

config.ImplementationType = typeof(MyInbox); //Your Inbox class

});

});

````

## See Also

\* [Local Event Bus](Local-Event-Bus.md)

#### Azure Service Bus Integration

# Distributed Event Bus Azure Integration

> This document explains **\*\*how to configure the [**Azure Service Bus**](https://azure.microsoft.com/en-us/services/service-bus/)\*\*** as the distributed event bus provider. See the [distributed event bus document](Distributed-Event-Bus.md) to learn how to use the distributed event bus system

## Installation

Use the ABP CLI to add [Volo.Abp.EventBus.Azure](https://www.nuget.org/packages/Volo.Abp.EventBus.Azure) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.EventBus.Azure` package.

\* Run `abp add-package Volo.Abp.EventBus.Azure` command.

If you want to do it manually, install the [Volo.Abp.EventBus.Azure](https://www.nuget.org/packages/Volo.Abp.EventBus.Azure) NuGet package to your project and add `[DependsOn(typeof(AbpEventBusAzureModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

You can configure using the standard [configuration system](Configuration.md), like using the `appsettings.json` file, or using the [options](Options.md) classes.

### `appsettings.json` file configuration

This is the simplest way to configure the Azure Service Bus settings. It is also very strong since you can use any other configuration source (like environment variables) that is [supported by the AspNet Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/).

**\*\*Example: The minimal configuration to connect to Azure Service Bus Namespace with default configurations\*\***

````json

{

"Azure": {

"ServiceBus": {

"Connections": {

"Default": {

"ConnectionString": "Endpoint=sb://sb-my-app.servicebus.windows.net/;SharedAccessKeyName={%{{{Policy Name}}}%};SharedAccessKey={};EntityPath=marketing-consent"

}

}

},

"EventBus": {

"ConnectionName": "Default",

"SubscriberName": "MySubscriberName",

"TopicName": "MyTopicName"

}

}

}

````

\* `MySubscriberName` is the name of this subscription, which is used as the **\*\*Subscriber\*\*** on the Azure Service Bus.

\* `MyTopicName` is the **\*\*topic name\*\***.

See [the Azure Service Bus document](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions) to understand these options better.

#### Connections

If you need to connect to another Azure Service Bus Namespace the Default, you need to configure the connection properties.

**\*\*Example: Declare two connections and use one of them for the event bus\*\***

````json

{

"Azure": {

"ServiceBus": {

"Connections": {

"Default": {

"ConnectionString": "Endpoint=sb://sb-my-app.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey={%{{{SharedAccessKey}}}%}"

},

"SecondConnection": {

"ConnectionString": "Endpoint=sb://sb-my-app.servicebus.windows.net/;SharedAccessKeyName={%{{{Policy Name}}}%};SharedAccessKey={%{{{SharedAccessKey}}}%}"

}

}

},

"EventBus": {

"ConnectionName": "SecondConnection",

"SubscriberName": "MySubscriberName",

"TopicName": "MyTopicName"

}

}

}

````

This allows you to use multiple Azure Service Bus namespaces in your application, but select one of them for the event bus.

You can use any of the [ServiceBusAdministrationClientOptions](https://docs.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.administration.servicebusadministrationclientoptions?view=azure-dotnet), [ServiceBusClientOptions](https://docs.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.servicebusclientoptions?view=azure-dotnet), [ServiceBusProcessorOptions](https://docs.microsoft.com/en-us/dotnet/api/azure.messaging.servicebus.servicebusprocessoroptions?view=azure-dotnet) properties for the connection.

**\*\*Example: Specify the Admin, Client and Processor options\*\***

````json

{

"Azure": {

"ServiceBus": {

"Connections": {

"Default": {

"ConnectionString": "Endpoint=sb://sb-my-app.servicebus.windows.net/;SharedAccessKeyName={%{{{Policy Name}}}%};SharedAccessKey={};EntityPath=marketing-consent",

"Admin": {

"Retry": {

"MaxRetries": 3

}

},

"Client": {

"RetryOptions": {

"MaxRetries": 1

}

},

"Processor": {

"AutoCompleteMessages": true,

"ReceiveMode": "ReceiveAndDelete"

}

}

}

},

"EventBus": {

"ConnectionName": "Default",

"SubscriberName": "MySubscriberName",

"TopicName": "MyTopicName"

}

}

}

````

### The Options Classes

`AbpAzureServiceBusOptions` and `AbpAzureEventBusOptions` classes can be used to configure the connection strings and event bus options for Azure Service Bus.

You can configure this options inside the `ConfigureServices` of your [module](Module-Development-Basics.md).

**\*\*Example: Configure the connection\*\***

````csharp

Configure<AbpAzureServiceBusOptions>(options =>

{

options.Connections.Default.ConnectionString = "Endpoint=sb://sb-my-app.servicebus.windows.net/;SharedAccessKeyName={%{{{Policy Name}}}%};SharedAccessKey={}";

options.Connections.Default.Admin.Retry.MaxRetries = 3;

options.Connections.Default.Client.RetryOptions.MaxRetries = 1;

});

````

Using these options classes can be combined with the `appsettings.json` way. Configuring an option property in the code overrides the value in the configuration file.

#### RabbitMQ Integration

# Distributed Event Bus RabbitMQ Integration

> This document explains **\*\*how to configure the [**RabbitMQ**](https://www.rabbitmq.com/)\*\*** as the distributed event bus provider. See the [distributed event bus document](Distributed-Event-Bus.md) to learn how to use the distributed event bus system

## Installation

Use the ABP CLI to add [Volo.Abp.EventBus.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.EventBus.RabbitMQ) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.EventBus.RabbitMQ` package.

\* Run `abp add-package Volo.Abp.EventBus.RabbitMQ` command.

If you want to do it manually, install the [Volo.Abp.EventBus.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.EventBus.RabbitMQ) NuGet package to your project and add `[DependsOn(typeof(AbpEventBusRabbitMqModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

You can configure using the standard [configuration system](Configuration.md), like using the `appsettings.json` file, or using the [options](Options.md) classes.

### `appsettings.json` file configuration

This is the simplest way to configure the RabbitMQ settings. It is also very strong since you can use any other configuration source (like environment variables) that is [supported by the AspNet Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/).

**\*\*Example: The minimal configuration to connect to a local RabbitMQ server with default configurations\*\***

````json

{

"RabbitMQ": {

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

````

\* `ClientName` is the name of this application, which is used as the **\*\*queue name\*\*** on the RabbitMQ.

\* `ExchangeName` is the **\*\*exchange name\*\***.

See [the RabbitMQ document](https://www.rabbitmq.com/dotnet-api-guide.html#exchanges-and-queues) to understand these options better.

#### Connections

If you need to connect to another server than the localhost, you need to configure the connection properties.

**\*\*Example: Specify the host name (as an IP address)\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

````

Defining multiple connections is allowed. In this case, you can specify the connection that is used for the event bus.

**\*\*Example: Declare two connections and use one of them for the event bus\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

},

"SecondConnection": {

"HostName": "321.321.321.321"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName",

"ConnectionName": "SecondConnection"

}

}

}

````

This allows you to use multiple RabbitMQ server in your application, but select one of them for the event bus.

You can use any of the [ConnectionFactory](http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.ConnectionFactory.html#properties) properties as the connection properties.

**\*\*Example: Specify the connection port\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123",

"Port": "5672"

}

}

}

}

````

If you need to connect to the RabbitMQ cluster, you can use the `;` character to separate the host names.

**\*\*Example: Connect to the RabbitMQ cluster\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123;234.234.234.234"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

```

### The Options Classes

`AbpRabbitMqOptions` and `AbpRabbitMqEventBusOptions` classes can be used to configure the connection strings and event bus options for the RabbitMQ.

You can configure this options inside the `ConfigureServices` of your [module](Module-Development-Basics.md).

**\*\*Example: Configure the connection\*\***

````csharp

Configure<AbpRabbitMqOptions>(options =>

{

options.Connections.Default.UserName = "user";

options.Connections.Default.Password = "pass";

options.Connections.Default.HostName = "123.123.123.123";

options.Connections.Default.Port = 5672;

});

````

**\*\*Example: Configure the client, exchange names and prefetchCount\*\***

````csharp

Configure<AbpRabbitMqEventBusOptions>(options =>

{

options.ClientName = "TestApp1";

options.ExchangeName = "TestMessages";

options.PrefetchCount = 1;

});

````

Using these options classes can be combined with the `appsettings.json` way. Configuring an option property in the code overrides the value in the configuration file.

#### Kafka Integration

# Distributed Event Bus Kafka Integration

> This document explains **\*\*how to configure the [**Kafka**](https://kafka.apache.org/)\*\*** as the distributed event bus provider. See the [distributed event bus document](Distributed-Event-Bus.md) to learn how to use the distributed event bus system

## Installation

Use the ABP CLI to add [Volo.Abp.EventBus.Kafka](https://www.nuget.org/packages/Volo.Abp.EventBus.Kafka) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.EventBus.Kafka` package.

\* Run `abp add-package Volo.Abp.EventBus.Kafka` command.

If you want to do it manually, install the [Volo.Abp.EventBus.Kafka](https://www.nuget.org/packages/Volo.Abp.EventBus.Kafka) NuGet package to your project and add `[DependsOn(typeof(AbpEventBusKafkaModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

You can configure using the standard [configuration system](Configuration.md), like using the `appsettings.json` file, or using the [options](Options.md) classes.

### `appsettings.json` file configuration

This is the simplest way to configure the Kafka settings. It is also very strong since you can use any other configuration source (like environment variables) that is [supported by the AspNet Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/).

**\*\*Example: The minimal configuration to connect to a local kafka server with default configurations\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "localhost:9092"

}

},

"EventBus": {

"GroupId": "MyGroupId",

"TopicName": "MyTopicName"

}

}

}

````

\* `MyGroupId` is the name of this application, which is used as the **\*\*GroupId\*\*** on the Kakfa.

\* `MyTopicName` is the **\*\*topic name\*\***.

See [the Kafka document](https://docs.confluent.io/current/clients/confluent-kafka-dotnet/api/Confluent.Kafka.html) to understand these options better.

#### Connections

If you need to connect to another server than the localhost, you need to configure the connection properties.

**\*\*Example: Specify the host name (as an IP address)\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "123.123.123.123:9092"

}

},

"EventBus": {

"GroupId": "MyGroupId",

"TopicName": "MyTopicName"

}

}

}

````

Defining multiple connections is allowed. In this case, you can specify the connection that is used for the event bus.

**\*\*Example: Declare two connections and use one of them for the event bus\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "123.123.123.123:9092"

},

"SecondConnection": {

"BootstrapServers": "321.321.321.321:9092"

}

},

"EventBus": {

"GroupId": "MyGroupId",

"TopicName": "MyTopicName",

"ConnectionName": "SecondConnection"

}

}

}

````

This allows you to use multiple Kafka cluster in your application, but select one of them for the event bus.

You can use any of the [ClientConfig](https://docs.confluent.io/current/clients/confluent-kafka-dotnet/api/Confluent.Kafka.ClientConfig.html) properties as the connection properties.

**\*\*Example: Specify the socket timeout\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "123.123.123.123:9092",

"SocketTimeoutMs": 60000

}

}

}

}

````

### The Options Classes

`AbpKafkaOptions` and `AbpKafkaEventBusOptions` classes can be used to configure the connection strings and event bus options for the Kafka.

You can configure this options inside the `ConfigureServices` of your [module](Module-Development-Basics.md).

**\*\*Example: Configure the connection\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.Connections.Default.BootstrapServers = "123.123.123.123:9092";

options.Connections.Default.SaslUsername = "user";

options.Connections.Default.SaslPassword = "pwd";

});

````

**\*\*Example: Configure the consumer config\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.ConfigureConsumer = config =>

{

config.GroupId = "MyGroupId";

config.EnableAutoCommit = false;

};

});

````

**\*\*Example: Configure the producer config\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.ConfigureProducer = config =>

{

config.MessageTimeoutMs = 6000;

config.Acks = Acks.All;

};

});

````

**\*\*Example: Configure the topic specification\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.ConfigureTopic = specification =>

{

specification.ReplicationFactor = 3;

specification.NumPartitions = 3;

};

});

````

Using these options classes can be combined with the `appsettings.json` way. Configuring an option property in the code overrides the value in the configuration file.

#### Rebus Integration

# Distributed Event Bus Rebus Integration

> This document explains **\*\*how to configure the [**Rebus**](http://mookid.dk/category/rebus/)\*\*** as the distributed event bus provider. See the [distributed event bus document](Distributed-Event-Bus.md) to learn how to use the distributed event bus system

## Installation

Use the ABP CLI to add [Volo.Abp.EventBus.Rebus](https://www.nuget.org/packages/Volo.Abp.EventBus.Rebus) NuGet package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) if you haven't installed before.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.EventBus.Rebus` package.

\* Run `abp add-package Volo.Abp.EventBus.Rebus` command.

If you want to do it manually, install the [Volo.Abp.EventBus.Rebus](https://www.nuget.org/packages/Volo.Abp.EventBus.Rebus) NuGet package to your project and add `[DependsOn(typeof(AbpEventBusRebusModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Configuration

You can configure using the standard [configuration system](Configuration.md), like using the [options](Options.md) classes.

### The Options Classes

`AbpRebusEventBusOptions` class can be used to configure the event bus options for the Rebus.

You can configure this options inside the `PreConfigureServices` of your [module](Module-Development-Basics.md).

**\*\*Example: Minimize configuration\*\***

```csharp

PreConfigure<AbpRebusEventBusOptions>(options =>

{

options.InputQueueName = "eventbus";

});

```

Rebus has many options, you can use the `Configurer` property of `AbpRebusEventBusOptions` class to configure.

Default events are **\*\*stored in memory\*\***. See the [rebus document](https://github.com/rebus-org/Rebus/wiki/Transport) for more details.

**\*\*Example: Configure the store\*\***

````csharp

PreConfigure<AbpRebusEventBusOptions>(options =>

{

options.InputQueueName = "eventbus";

options.Configurer = rebusConfigurer =>

{

rebusConfigurer.Transport(t => t.UseMsmq("eventbus"));

rebusConfigurer.Subscriptions(s => s.UseJsonFile(@"subscriptions.json"));

};

});

````

You can use the `Publish` properpty of `AbpRebusEventBusOptions` class to change the publishing method

**\*\*Example: Configure the event publishing\*\***

````csharp

PreConfigure<AbpRebusEventBusOptions>(options =>

{

options.InputQueueName = "eventbus";

options.Publish = async (bus, type, data) =>

{

await bus.Publish(data);

};

});

````

## Features

# Features

ABP Feature system is used to **\*\*enable\*\***, **\*\*disable\*\*** or **\*\*change the behavior\*\*** of the application features **\*\*on runtime\*\***.

The runtime value for a feature is generally a `boolean` value, like `true` (enabled) or `false` (disabled). However, you can get/set **\*\*any kind\*\*** of value for feature.

Feature system was originally designed to control the tenant features in a **\*\*[**multi-tenant**](Multi-Tenancy.md)\*\*** application. However, it is **\*\*extensible\*\*** and capable of determining the features by any condition.

> The feature system is implemented with the [Volo.Abp.Features](https://www.nuget.org/packages/Volo.Abp.Features) NuGet package. Most of the times you don't need to manually [install it](https://abp.io/package-detail/Volo.Abp.Features) since it comes pre-installed with the [application startup template](Startup-Templates/Application.md).

## Checking for the Features

Before explaining to define features, let's see how to check a feature value in your application code.

### RequiresFeature Attribute

`[RequiresFeature]` attribute (defined in the `Volo.Abp.Features` namespace) is used to declaratively check if a feature is `true` (enabled) or not. It is a useful shortcut for the `boolean` features.

**\*\*Example: Check if the "PDF Reporting" feature enabled\*\***

```csharp

public class ReportingAppService : ApplicationService, IReportingAppService

{

[RequiresFeature("MyApp.PdfReporting")]

public async Task<PdfReportResultDto> GetPdfReportAsync()

{

//TODO...

}

}

```

\* `RequiresFeature(...)` simply gets a feature name to check if it is enabled or not. If not enabled, an authorization [exception](Exception-Handling.md) is thrown and a proper response is returned to the client side.

\* `[RequiresFeature]` can be used for a **\*\*method\*\*** or a **\*\*class\*\***. When you use it for a class, all the methods of that class require the given feature.

\* `RequiresFeature` may get multiple feature names, like `[RequiresFeature("Feature1", "Feature2")]`. In this case ABP checks if any of the features enabled. Use `RequiresAll` option, like `[RequiresFeature("Feature1", "Feature2", RequiresAll = true)]` to force to check all of the features to be enabled.

\* Multiple usage of `[RequiresFeature]` attribute is supported for a method or class. ABP checks all of them in that case.

> Feature name can be any arbitrary string. It should be unique for a feature.

#### About the Interception

ABP Framework uses the interception system to make the `[RequiresFeature]` attribute working. So, it can work with any class (application services, controllers...) that is injected from the [dependency injection](Dependency-Injection.md).

However, there are **\*\*some rules should be followed\*\*** in order to make it working;

\* If you are **\*\*not injecting\*\*** the service over an interface (like `IMyService`), then the methods of the service must be `virtual`. Otherwise, [dynamic proxy / interception](Dynamic-Proxying-Interceptors.md) system can not work.

\* Only `async` methods (methods returning a `Task` or `Task<T>`) are intercepted.

> There is an exception for the **\*\*controller and razor page methods\*\***. They **\*\*don't require\*\*** the following the rules above, since ABP Framework uses the action/page filters to implement the feature checking in this case.

### IFeatureChecker Service

`IFeatureChecker` allows to check a feature in your application code.

#### IsEnabledAsync

Returns `true` if the given feature is enabled. So, you can conditionally execute your business flow.

**\*\*Example: Check if the "PDF Reporting" feature enabled\*\***

```csharp

public class ReportingAppService : ApplicationService, IReportingAppService

{

private readonly IFeatureChecker \_featureChecker;

public ReportingAppService(IFeatureChecker featureChecker)

{

\_featureChecker = featureChecker;

}

public async Task<PdfReportResultDto> GetPdfReportAsync()

{

if (await \_featureChecker.IsEnabledAsync("MyApp.PdfReporting"))

{

//TODO...

}

else

{

//TODO...

}

}

}

```

`IsEnabledAsync` has overloads to check multiple features in one method call.

#### GetOrNullAsync

Gets the current value for a feature. This method returns a `string`, so you store any kind of value inside it, by converting to or from `string`.

**\*\*Example: Check the maximum product count allowed\*\***

```csharp

public class ProductController : AbpController

{

private readonly IFeatureChecker \_featureChecker;

public ProductController(IFeatureChecker featureChecker)

{

\_featureChecker = featureChecker;

}

public async Task<IActionResult> Create(CreateProductModel model)

{

var currentProductCount = await GetCurrentProductCountFromDatabase();

//GET THE FEATURE VALUE

var maxProductCountLimit =

await \_featureChecker.GetOrNullAsync("MyApp.MaxProductCount");

if (currentProductCount >= Convert.ToInt32(maxProductCountLimit))

{

throw new BusinessException(

"MyApp:ReachToMaxProductCountLimit",

$"You can not create more than {maxProductCountLimit} products!"

);

}

//TODO: Create the product in the database...

}

private async Task<int> GetCurrentProductCountFromDatabase()

{

throw new System.NotImplementedException();

}

}

```

This example uses a numeric value as a feature limit product counts for a user/tenant in a SaaS application.

Instead of manually converting the value to `int`, you can use the generic overload of the `GetAsync` method:

```csharp

var maxProductCountLimit = await \_featureChecker.GetAsync<int>("MyApp.MaxProductCount");

```

#### Extension Methods

There are some useful extension methods for the `IFeatureChecker` interface;

\* `Task<T> GetAsync<T>(string name, T defaultValue = default)`: Used to get a value of a feature with the given type `T`. Allows to specify a `defaultValue` that is returned when the feature value is `null`.

\* `CheckEnabledAsync(string name)`: Checks if given feature is enabled. Throws an `AbpAuthorizationException` if the feature was not `true` (enabled).

## Defining the Features

A feature should be defined to be able to check it.

### FeatureDefinitionProvider

Create a class inheriting the `FeatureDefinitionProvider` to define features.

**\*\*Example: Defining features\*\***

```csharp

using Volo.Abp.Features;

namespace FeaturesDemo

{

public class MyFeatureDefinitionProvider : FeatureDefinitionProvider

{

public override void Define(IFeatureDefinitionContext context)

{

var myGroup = context.AddGroup("MyApp");

myGroup.AddFeature("MyApp.PdfReporting", defaultValue: "false");

myGroup.AddFeature("MyApp.MaxProductCount", defaultValue: "10");

}

}

}

```

> ABP automatically discovers this class and registers the features. No additional configuration required.

> This class is generally created in the `Application.Contracts` project of your solution.

\* In the `Define` method, you first need to add a **\*\*feature group\*\*** for your application/module or get an existing group then add **\*\*features\*\*** to this group.

\* First feature, named `MyApp.PdfReporting`, is a `boolean` feature with `false` as the default value.

\* Second feature, named `MyApp.MaxProductCount`, is a numeric feature with `10` as the default value.

Default value is used if there is no other value set for the current user/tenant.

### Other Feature Properties

While these minimal definitions are enough to make the feature system working, you can specify the **\*\*optional properties\*\*** for the features;

\* `DisplayName`: A localizable string that will be used to show the feature name on the user interface.

\* `Description`: A longer localizable text to describe the feature.

\* `ValueType`: Type of the feature value. Can be a class implementing the `IStringValueType`. Built-in types:

\* `ToggleStringValueType`: Used to define `true`/`false`, `on`/`off`, `enabled`/`disabled` style features. A checkbox is shown on the UI.

\* `FreeTextStringValueType`: Used to define free text values. A textbox is shown on the UI.

\* `SelectionStringValueType`: Used to force the value to be selected from a list. A dropdown list is shown on the UI.

\* `IsVisibleToClients` (default: `true`): Set false to hide the value of this feature from clients (browsers). Sharing the value with the clients helps them to conditionally show/hide/change the UI parts based on the feature value.

\* `Properties`: A dictionary to set/get arbitrary key-value pairs related to this feature. This can be a point for customization.

So, based on these descriptions, it would be better to define these features as shown below:

```csharp

using FeaturesDemo.Localization;

using Volo.Abp.Features;

using Volo.Abp.Localization;

using Volo.Abp.Validation.StringValues;

namespace FeaturesDemo

{

public class MyFeatureDefinitionProvider : FeatureDefinitionProvider

{

public override void Define(IFeatureDefinitionContext context)

{

var myGroup = context.AddGroup("MyApp");

myGroup.AddFeature(

"MyApp.PdfReporting",

defaultValue: "false",

displayName: LocalizableString

.Create<FeaturesDemoResource>("PdfReporting"),

valueType: new ToggleStringValueType()

);

myGroup.AddFeature(

"MyApp.MaxProductCount",

defaultValue: "10",

displayName: LocalizableString

.Create<FeaturesDemoResource>("MaxProductCount"),

valueType: new FreeTextStringValueType(

new NumericValueValidator(0, 1000000))

);

}

}

}

```

\* `FeaturesDemoResource` is the project name in this example code. See the [localization document](Localization.md) for details about the localization system.

\* First feature is set to `ToggleStringValueType`, while the second one is set to `FreeTextStringValueType` with a numeric validator that allows to the values from `0` to `1,000,000`.

Remember to define the localization the keys in your localization file:

````json

"PdfReporting": "PDF Reporting",

"MaxProductCount": "Maximum number of products"

````

See the [localization document](Localization.md) for details about the localization system.

### Feature Management Modal

The [application startup template](Startup-Templates/Application.md) comes with the [tenant management](Modules/Tenant-Management.md) and the [feature management](Modules/Feature-Management.md) modules pre-installed.

Whenever you define a new feature, it will be available on the **\*\*feature management modal\*\***. To open this modal, navigate to the **\*\*tenant management page\*\*** and select the `Features` action for a tenant (create a new tenant if there is no tenant yet):

![features-action](images/features-action.png)

This action opens a modal to manage the feature values for the selected tenant:

![features-modal](images/features-modal.png)

So, you can enable, disable and set values for a tenant. These values will be used whenever a user of this tenant uses the application.

See the *\*Feature Management\** section below to learn more about managing the features.

### Child Features

A feature may have child features. This is especially useful if you want to create a feature that is selectable only if another feature was enabled.

**\*\*Example: Defining child features\*\***

```csharp

using FeaturesDemo.Localization;

using Volo.Abp.Features;

using Volo.Abp.Localization;

using Volo.Abp.Validation.StringValues;

namespace FeaturesDemo

{

public class MyFeatureDefinitionProvider : FeatureDefinitionProvider

{

public override void Define(IFeatureDefinitionContext context)

{

var myGroup = context.AddGroup("MyApp");

var reportingFeature = myGroup.AddFeature(

"MyApp.Reporting",

defaultValue: "false",

displayName: LocalizableString

.Create<FeaturesDemoResource>("Reporting"),

valueType: new ToggleStringValueType()

);

reportingFeature.CreateChild(

"MyApp.PdfReporting",

defaultValue: "false",

displayName: LocalizableString

.Create<FeaturesDemoResource>("PdfReporting"),

valueType: new ToggleStringValueType()

);

reportingFeature.CreateChild(

"MyApp.ExcelReporting",

defaultValue: "false",

displayName: LocalizableString

.Create<FeaturesDemoResource>("ExcelReporting"),

valueType: new ToggleStringValueType()

);

}

}

}

```

The example above defines a *\*Reporting\** feature with two children: *\*PDF Reporting\** and *\*Excel Reporting\**.

### Changing Features Definitions of a Depended Module

A class deriving from the `FeatureDefinitionProvider` (just like the example above) can also get the existing feature definitions (defined by the depended [modules](Module-Development-Basics.md)) and change their definitions.

**\*\*Example: Manipulate an existing feature definition\*\***

```csharp

var someGroup = context.GetGroupOrNull("SomeModule");

var feature = someGroup.Features.FirstOrDefault(f => f.Name == "SomeFeature");

if (feature != null)

{

feature.Description = ...

feature.CreateChild(...);

}

```

## Check a Feature in the Client Side

A feature value is available at the client side too, unless you set `IsVisibleToClients` to `false` on the feature definition. The feature values are exposed from the [Application Configuration API](API/Application-Configuration.md) and usable via some services on the UI.

See the following documents to learn how to check features in different UI types:

\* [ASP.NET Core MVC / Razor Pages / JavaScript API](UI/AspNetCore/JavaScript-API/Features.md)

\* [Angular](UI/Angular/Features.md)

**\*\*Blazor\*\*** applications can use the same `IFeatureChecker` service as explained above.

## Feature Management

Feature management is normally done by an admin user using the feature management modal:

![features-modal](images/features-modal.png)

This modal is available on the related entities, like tenants in a multi-tenant application. To open it, navigate to the **\*\*Tenant Management\*\*** page (for a multi-tenant application), click to the **\*\*Actions\*\*** button left to the Tenant and select the **\*\*Features\*\*** action.

If you need to manage features by code, inject the `IFeatureManager` service.

**\*\*Example: Enable PDF reporting for a tenant\*\***

```csharp

public class MyService : ITransientDependency

{

private readonly IFeatureManager \_featureManager;

public MyService(IFeatureManager featureManager)

{

\_featureManager = featureManager;

}

public async Task EnablePdfReporting(Guid tenantId)

{

await \_featureManager.SetForTenantAsync(

tenantId,

"MyApp.PdfReporting",

true.ToString()

);

}

}

```

`IFeatureManager` is defined by the Feature Management module. It comes pre-installed with the application startup template. See the [feature management module documentation](Modules/Feature-Management.md) for more information.

## Advanced Topics

### Feature Value Providers

Feature system is extensible. Any class derived from `FeatureValueProvider` (or implements `IFeatureValueProvider`) can contribute to the feature system. A value provider is responsible to **\*\*obtain the current value\*\*** of a given feature.

Feature value providers are **\*\*executed one by one\*\***. If one of them return a non-null value, then this feature value is used and the other providers are not executed.

There are three pre-defined value providers, executed by the given order:

\* `TenantFeatureValueProvider` tries to get if the feature value is explicitly set for the **\*\*current tenant\*\***.

\* `EditionFeatureValueProvider` tries to get the feature value for the current edition. Edition Id is obtained from the current principal identity (`ICurrentPrincipalAccessor`) with the claim name `editionid` (a constant defined as`AbpClaimTypes.EditionId`). Editions are not implemented for the [tenant management](Modules/Tenant-Management.md) module. You can implement it yourself or consider to use the [SaaS module](https://commercial.abp.io/modules/Volo.Saas) of the ABP Commercial.

\* `DefaultValueFeatureValueProvider` gets the default value of the feature.

You can write your own provider by inheriting the `FeatureValueProvider`.

**\*\*Example: Enable all features for a user with "SystemAdmin" as a "User\_Type" claim value\*\***

```csharp

using System.Threading.Tasks;

using Volo.Abp.Features;

using Volo.Abp.Security.Claims;

using Volo.Abp.Validation.StringValues;

namespace FeaturesDemo

{

public class SystemAdminFeatureValueProvider : FeatureValueProvider

{

public override string Name => "SA";

private readonly ICurrentPrincipalAccessor \_currentPrincipalAccessor;

public SystemAdminFeatureValueProvider(

IFeatureStore featureStore,

ICurrentPrincipalAccessor currentPrincipalAccessor)

: base(featureStore)

{

\_currentPrincipalAccessor = currentPrincipalAccessor;

}

public override Task<string> GetOrNullAsync(FeatureDefinition feature)

{

if (feature.ValueType is ToggleStringValueType &&

\_currentPrincipalAccessor.Principal?.FindFirst("User\_Type")?.Value == "SystemAdmin")

{

return Task.FromResult("true");

}

return null;

}

}

}

```

If a provider returns `null`, then the next provider is executed.

Once a provider is defined, it should be added to the `AbpFeatureOptions` as shown below:

```csharp

Configure<AbpFeatureOptions>(options =>

{

options.ValueProviders.Add<SystemAdminFeatureValueProvider>();

});

```

Use this code inside the `ConfigureServices` of your [module](Module-Development-Basics.md) class.

### Feature Store

`IFeatureStore` is the only interface that needs to be implemented to read the value of features from a persistence source, generally a database system. The Feature Management module implements it and pre-installed in the application startup template. See the [feature management module documentation](https://docs.abp.io/en/abp/latest/Modules/Feature-Management) for more information

## Global Features

# Global Features

Global Feature system is used to enable/disable an application feature on development time. It is done on the development time, because some **\*\*services\*\*** (e.g. controllers) are removed from the application model and **\*\*database tables\*\*** are not created for the disabled features, which is not possible on runtime.

Global Features system is especially useful if you want to develop a reusable application module with optional features. If the final application doesn't want to use some of the features, it can disable these features.

> If you are looking for a system to enable/disable features based on current tenant or any other condition, please see the [Features](Features.md) document.

## Installation

> This package is already installed by default with the startup template. So, most of the time, you don't need to install it manually.

Open a command line window in the folder of the project (.csproj file) and type the following command:

```bash

abp add-package Volo.Abp.GlobalFeatures

```

## Defining a Global Feature

A feature class is something like that:

```csharp

[GlobalFeatureName("Shopping.Payment")]

public class PaymentFeature

{

}

```

## Enable/Disable Global Features

Use `GlobalFeatureManager.Instance` to enable/disable a global feature.

```csharp

// Able to Enable/Disable with generic type parameter.

GlobalFeatureManager.Instance.Enable<PaymentFeature>();

GlobalFeatureManager.Instance.Disable<PaymentFeature>();

// Also able to Enable/Disable with string feature name.

GlobalFeatureManager.Instance.Enable("Shopping.Payment");

GlobalFeatureManager.Instance.Disable("Shopping.Payment");

```

> Global Features are disabled unless they are explicitly enabled.

### Where to Configure Global Features?

Global Features have to be configured before application startup. Since the `GlobalFeatureManager.Instance` is a singleton object, one-time, static configuration is enough. It is suggested to enable/disable global features in `PreConfigureServices` method of your module. You can use the `OneTimeRunner` utility class to make sure it runs only once:

```csharp

private static readonly OneTimeRunner OneTimeRunner = new OneTimeRunner();

public override void PreConfigureServices(ServiceConfigurationContext context)

{

OneTimeRunner.Run(() =>

{

GlobalFeatureManager.Instance.Enable<PaymentFeature>();

});

}

```

## Check for a Global Feature

```csharp

GlobalFeatureManager.Instance.IsEnabled<PaymentFeature>()

GlobalFeatureManager.Instance.IsEnabled("Shopping.Payment")

```

Both methods return `bool`. So, you can write conditional logic as shown below:

```csharp

if (GlobalFeatureManager.Instance.IsEnabled<PaymentFeature>())

{

// Some strong payment codes here...

}

```

### RequiresGlobalFeature Attribute

Beside the manual check, there is `[RequiresGlobalFeature]` attribute to check it declaratively for a controller or page. ABP returns HTTP Response `404` if the related feature was disabled.

```csharp

[RequiresGlobalFeature(typeof(PaymentFeature))]

public class PaymentController : AbpController

{

}

```

## Grouping Features of a Module

It is common to group global features of a module to allow the final application developer easily discover and configure the features. Following example shows how to group features of a module.

Assume that we've defined a global feature for `Subscription` feature of an `Ecommerce` module:

```csharp

[GlobalFeatureName("Ecommerce.Subscription")]

public class SubscriptionFeature : GlobalFeature

{

public SubscriptionFeature(GlobalModuleFeatures module)

: base(module)

{

}

}

```

You can define as many features as you need in your module. Then define a class to group these features together:

```csharp

public class GlobalEcommerceFeatures : GlobalModuleFeatures

{

public const string ModuleName = "Ecommerce";

public SubscriptionFeature Subscription => GetFeature<SubscriptionFeature>();

public GlobalEcommerceFeatures(GlobalFeatureManager featureManager)

: base(featureManager)

{

AddFeature(new SubscriptionFeature(this));

}

}

```

Finally, you can create an extension method on `GlobalModuleFeaturesDictionary`:

```csharp

public static class GlobalModuleFeaturesDictionaryEcommerceExtensions

{

public static GlobalEcommerceFeatures Ecommerce(

this GlobalModuleFeaturesDictionary modules)

{

return modules.GetOrAdd(

GlobalEcommerceFeatures.ModuleName,

\_ => new GlobalEcommerceFeatures(modules.FeatureManager)

) as GlobalEcommerceFeatures;

}

```

Then `GlobalFeatureManager.Instance.Modules.Ecommerce()` can be used to access the global features of your module. Examples usages:

```csharp

GlobalFeatureManager.Instance.Modules.Ecommerce().Subscription.Enable();

GlobalFeatureManager.Instance.Modules.Ecommerce().EnableAll();

```

## GUID Generation

# GUID Generation

GUID is a common **\*\*primary key type\*\*** that is used in database management systems. ABP Framework prefers GUID as the primary for pre-built [application modules](Modules/Index.md). Also, `ICurrentUser.Id` property ([see](CurrentUser.md)) is type of GUID, that means the ABP Framework assumes that the User Id is always GUID.

## Why Prefer GUID?

GUID has advantages and disadvantages. You can find many articles on the web related to this topic, so we will not discuss all again, but will list the most fundamental advantages:

\* It is **\*\*usable\*\*** in all database providers.

\* It allows to **\*\*determine the primary key\*\*** on the client side, without needing a **\*\*database round trip\*\*** to generate the Id value. This can be more performant while inserting new records to the database and allows us to know the PK before interacting to the database.

\* GUIDs are **\*\*naturally unique\*\*** which has some advantages in the following situations if;

\* You need to integrate to **\*\*external\*\*** systems.

\* You need to **\*\*split or merge\*\*** different tables.

\* You are creating **\*\*distributed systems\*\***.

\* GUIDs are impossible to guess, so they can be **\*\*more secure\*\*** compared to auto-increment Id values in some cases.

While there are some disadvantages (just search it on the web), we found these advantages much more important while designing the ABP Framework.

## IGuidGenerator

The most important problem with GUID is that it is **\*\*not sequential by default\*\***. When you use the GUID as the primary key and set it as the **\*\*clustered index\*\*** (which is default) for your table, it brings a significant **\*\*performance problem on insert\*\*** (because inserting new record may need to re-order the existing records).

So, **\*\*never use** `Guid.NewGuid()` **to create Ids\*\*** for your entities!

One good solution to this problem is to generate **\*\*sequential GUIDs\*\***, which is provided by the ABP Framework out of the box. `IGuidGenerator` service creates sequential GUIDs (implemented by the `SequentialGuidGenerator` by default). Use `IGuidGenerator.Create()` when you need to manually set Id of an [entity](Entities.md).

**\*\*Example: An entity with GUID primary key and creating the entity\*\***

Assume that you've a `Product` [entity](Entities.md) that has a `Guid` key:

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace AbpDemo

{

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

private Product() { /\* This constructor is used by the ORM/database provider \*/ }

public Product(Guid id, string name)

: base(id)

{

Name = name;

}

}

}

````

And you want to create a new product:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Guids;

namespace AbpDemo

{

public class MyProductService : ITransientDependency

{

private readonly IRepository<Product, Guid> \_productRepository;

private readonly IGuidGenerator \_guidGenerator;

public MyProductService(

IRepository<Product, Guid> productRepository,

IGuidGenerator guidGenerator)

{

\_productRepository = productRepository;

\_guidGenerator = guidGenerator;

}

public async Task CreateAsync(string productName)

{

var product = new Product(\_guidGenerator.Create(), productName);

await \_productRepository.InsertAsync(product);

}

}

}

````

This service injects the `IGuidGenerator` in the constructor. If your class is an [application service](Application-Services.md) or deriving from one of the other base classes, you can directly use the `GuidGenerator` base property which is a pre-injected `IGuidGenerator` instance.

## Options

### AbpSequentialGuidGeneratorOptions

`AbpSequentialGuidGeneratorOptions` is the [option class](Options.md) that is used to configure the sequential GUID generation. It has a single property:

\* `DefaultSequentialGuidType` (`enum` of type `SequentialGuidType`): The strategy used while generating GUID values.

Database providers behaves differently while processing GUIDs, so you should set it based on your database provider. `SequentialGuidType` has the following `enum` members:

\* `SequentialAtEnd` (**\*\*default\*\***) works well with the [SQL Server](Entity-Framework-Core.md).

\* `SequentialAsString` is used by [MySQL](Entity-Framework-Core-MySQL.md) and [PostgreSQL](Entity-Framework-Core-PostgreSQL.md).

\* `SequentialAsBinary` is used by [Oracle](Entity-Framework-Core-Oracle.md).

Configure this option in the `ConfigureServices` method of your [module](Module-Development-Basics.md), as shown below:

````csharp

Configure<AbpSequentialGuidGeneratorOptions>(options =>

{

options.DefaultSequentialGuidType = SequentialGuidType.SequentialAsBinary;

});

````

> EF Core [integration packages](https://docs.abp.io/en/abp/latest/Entity-Framework-Core-Other-DBMS) sets this option to a proper value for the related DBMS. So, most of the times, you don't need to set this option if you are using these integration packages.

## Image Manipulation

# Image Manipulation

ABP Framework provides services to compress and resize images and implements these services with popular [ImageSharp](https://sixlabors.com/products/imagesharp/) and [Magick.NET](https://github.com/dlemstra/Magick.NET) libraries. You can use these services in your reusable modules, libraries and applications, so you don't depend on a specific imaging library.

> The image resizer/compressor system is designed to be extensible. You can implement your own image resizer/compressor contributor and use it in your application.

## Installation

You can add this package to your application by either using the [ABP CLI](CLI.md) or manually installing it. Using the [ABP CLI](CLI.md) is the recommended approach.

### Using the ABP CLI

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

```bash

abp add-package Volo.Abp.Imaging.Abstractions

```

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Imaging.Abstractions](https://www.nuget.org/packages/Volo.Abp.Imaging.Abstractions) NuGet package to your project:

```

Install-Package Volo.Abp.Imaging.Abstractions

```

2. Add the `AbpImagingAbstractionsModule` to the dependency list of your module:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpImagingAbstractionsModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

```

## Providers

ABP Framework provides two image resizer/compressor implementations out of the box:

\* [Magick.NET](#magick-net-provider)

\* [ImageSharp](#imagesharp-provider)

You should install one of these provides to make it actually working.

> If none of the provider packages installed into your application, compress/resize operations return the untouched input image.

## IImageResizer

You can [inject](Dependency-Injection.md) the `IImageResizer` service and use it for image resize operations. Here is the available methods of the `IImageResizer` service:

```csharp

public interface IImageResizer

{

/\* Works with a Stream object that represents an image \*/

Task<ImageResizeResult<Stream>> ResizeAsync(

Stream stream,

ImageResizeArgs resizeArgs,

string mimeType = null,

CancellationToken cancellationToken = default

);

/\* Works with a byte array that contains an image file \*/

Task<ImageResizeResult<byte[]>> ResizeAsync(

byte[] bytes,

ImageResizeArgs resizeArgs,

string mimeType = null,

CancellationToken cancellationToken = default

);

}

```

**\*\*Example usage:\*\***

```csharp

var result = await \_imageResizer.ResizeAsync(

stream, /\* A stream object that represents an image \*/

new ImageResizeArgs

{

Width = 100,

Height = 100,

Mode = ImageResizeMode.Crop

},

mimeType: "image/jpeg"

);

```

> You can use `MimeTypes.Image.Jpeg` constant instead of the `image/jpeg` magic string used in that example.

### ImageResizeArgs

The `ImageResizeArgs` is a class that is used to define the resize operation parameters. It has the following properties:

\* `Width`: The width of the resized image.

\* `Height`: The height of the resized image.

\* `Mode`: The resize mode (see the [ImageResizeMode](#imageresizemode) section for more information).

### ImageResizeMode

The `ImageResizeMode` is an enum that is used to define the resize mode. It has the following values:

```csharp

public enum ImageResizeMode : byte

{

None = 0,

Stretch = 1,

BoxPad = 2,

Min = 3,

Max = 4,

Crop = 5,

Pad = 6,

Default = 7

}

```

> See the [ImageSharp documentation](https://docs.sixlabors.com/api/ImageSharp/SixLabors.ImageSharp.Processing.ResizeMode.html) for more information about the resize modes.

### ImageResizeResult

The `ImageResizeResult` is a generic class that is used to return the result of the image resize operations. It has the following properties:

\* `Result`: The resized image (stream or byte array).

\* `State`: The result of the resize operation (type: `ImageProcessState`).

### ImageProcessState

The `ImageProcessState` is an enum that is used to return the the result of the image resize operations. It has the following values:

```csharp

public enum ImageProcessState : byte

{

Done = 1,

Canceled = 2,

Unsupported = 3,

}

```

### ImageResizeOptions

`ImageResizeOptions` is an [options object](Options.md) that is used to configure the image resize system. It has the following properties:

\* `DefaultResizeMode`: The default resize mode. (Default: `ImageResizeMode.None`)

## IImageCompressor

You can [inject](Dependency-Injection.md) the `IImageCompressor` service and use it for image compression operations. Here is the available methods of the `IImageCompressor` service:

```csharp

public interface IImageCompressor

{

/\* Works with a Stream object that represents an image \*/

Task<ImageCompressResult<Stream>> CompressAsync(

Stream stream,

string mimeType = null,

CancellationToken cancellationToken = default

);

/\* Works with a byte array that contains an image file \*/

Task<ImageCompressResult<byte[]>> CompressAsync(

byte[] bytes,

string mimeType = null,

CancellationToken cancellationToken = default

);

}

```

**\*\*Example usage:\*\***

```csharp

var result = await \_imageCompressor.CompressAsync(

stream, /\* A stream object that represents an image \*/

mimeType: "image/jpeg"

);

```

### ImageCompressResult

The `ImageCompressResult` is a generic class that is used to return the result of the image compression operations. It has the following properties:

\* `Result`: The compressed image (stream or byte array).

\* `State`: The result of the compress operation (type: `ImageProcessState`).

### ImageProcessState

The `ImageProcessState` is an enum that is used to return the the result of the image compress operations. It has the following values:

```csharp

public enum ImageProcessState : byte

{

Done = 1,

Canceled = 2,

Unsupported = 3,

}

```

## Magick.NET Provider

`Volo.Abp.Imaging.MagickNet` NuGet package implements the image operations using the [Magick.NET](https://github.com/dlemstra/Magick.NET) library.

## Installation

You can add this package to your application by either using the [ABP CLI](CLI.md) or manually installing it. Using the [ABP CLI](CLI.md) is the recommended approach.

### Using the ABP CLI

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

```bash

abp add-package Volo.Abp.Imaging.MagickNet

```

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Imaging.MagickNet](https://www.nuget.org/packages/Volo.Abp.Imaging.MagickNet) NuGet package to your project:

```

Install-Package Volo.Abp.Imaging.MagickNet

```

2. Add `AbpImagingMagickNetModule` to your [module](Module-Development-Basics.md)'s dependency list:

```csharp

[DependsOn(typeof(AbpImagingMagickNetModule))]

public class MyModule : AbpModule

{

//...

}

```

### Configuration

`MagickNetCompressOptions` is an [options object](Options.md) that is used to configure the Magick.NET image compression system. It has the following properties:

\* `OptimalCompression`: Indicates whether the optimal compression is enabled or not. (Default: `false`)

\* `IgnoreUnsupportedFormats`: Indicates whether the unsupported formats are ignored or not. (Default: `false`)

\* `Lossless`: Indicates whether the lossless compression is enabled or not. (Default: `false`)

## ImageSharp Provider

`Volo.Abp.Imaging.ImageSharp` NuGet package implements the image operations using the [ImageSharp](https://github.com/SixLabors/ImageSharp) library.

## Installation

You can add this package to your application by either using the [ABP CLI](CLI.md) or manually installing it. Using the [ABP CLI](CLI.md) is the recommended approach.

### Using the ABP CLI

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

```bash

abp add-package Volo.Abp.Imaging.ImageSharp

```

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Imaging.ImageSharp](https://www.nuget.org/packages/Volo.Abp.Imaging.ImageSharp) NuGet package to your project:

```

Install-Package Volo.Abp.Imaging.ImageSharp

```

2. Add `AbpImagingImageSharpModule` to your [module](Module-Development-Basics.md)'s dependency list:

```csharp

[DependsOn(typeof(AbpImagingImageSharpModule))]

public class MyModule : AbpModule

{

//...

}

```

### Configuration

`ImageSharpCompressOptions` is an [options object](Options.md) that is used to configure the ImageSharp image compression system. It has the following properties:

\* `DefaultQuality`: The default quality of the JPEG and WebP encoders. (Default: `75`)

\* [`JpegEncoder`](https://docs.sixlabors.com/api/ImageSharp/SixLabors.ImageSharp.Formats.Jpeg.JpegEncoder.html): The JPEG encoder. (Default: `JpegEncoder` with `Quality` set to `DefaultQuality`)

\* [`PngEncoder`](https://docs.sixlabors.com/api/ImageSharp/SixLabors.ImageSharp.Formats.Png.PngEncoder.html): The PNG encoder. (Default: `PngEncoder` with `IgnoreMetadata` set to `true` and `CompressionLevel` set to `PngCompressionLevel.BestCompression`)

\* [`WebPEncoder`](https://docs.sixlabors.com/api/ImageSharp/SixLabors.ImageSharp.Formats.Webp.WebpEncoder.html): The WebP encoder. (Default: `WebPEncoder` with `Quality` set to `DefaultQuality`)

**\*\*Example usage:\*\***

```csharp

Configure<ImageSharpCompressOptions>(options =>

{

options.JpegEncoder = new JpegEncoder

{

Quality = 60

};

options.PngEncoder = new PngEncoder

{

CompressionLevel = PngCompressionLevel.BestCompression

};

options.WebPEncoder = new WebPEncoder

{

Quality = 65

};

});

```

## ASP.NET Core Integration

`Volo.Abp.Imaging.AspNetCore` NuGet package defines attributes for controller actions that can automatically compress and/or resize uploaded files.

## Installation

You can add this package to your application by either using the [ABP CLI](CLI.md) or manually installing it. Using the [ABP CLI](CLI.md) is the recommended approach.

### Using the ABP CLI

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

```bash

abp add-package Volo.Abp.Imaging.AspNetCore

```

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Imaging.AspNetCore](https://www.nuget.org/packages/Volo.Abp.Imaging.AspNetCore) NuGet package to your project:

```

Install-Package Volo.Abp.Imaging.AspNetCore

```

2. Add `AbpImagingAspNetCoreModule` to your [module](Module-Development-Basics.md)'s dependency list:

```csharp

[DependsOn(typeof(AbpImagingAspNetCoreModule))]

public class MyModule : AbpModule

{

//...

}

```

### CompressImageAttribute

The `CompressImageAttribute` is used to compress the image before. `IFormFile`, `IRemoteStreamContent`, `Stream` and `IEnumrable<byte>` types are supported. It has the following properties:

\* `Parameters`: Names of the the parameters that are used to configure the image compression system. This is useful if your action has some non-image parameters. If you don't specify the parameters names, all of the method parameters are considered as image.

**\*\*Example usage:\*\***

```csharp

[HttpPost]

[CompressImage] /\* Compresses the given file (automatically determines the file mime type) \*/

public async Task<IActionResult> Upload(IFormFile file)

{

//...

}

```

### ResizeImageAttribute

The `ResizeImageAttribute` is used to resize the image before requesting the action. `IFormFile`, `IRemoteStreamContent`, `Stream` and `IEnumrable<byte>` types are supported. It has the following properties:

\* `Parameters`: Names of the the parameters that are used to configure the image resize system. This is useful if your action has some non-image parameters. If you don't specify the parameters names, all of the method parameters are considered as image.

\* `Width`: Target width of the resized image.

\* `Height`: Target height of the resized image.

\* `Mode`: The resize mode (see the [ImageResizeMode](#imageresizemode) section for more information).

**\*\*Example usage:\*\***

```csharp

[HttpPost]

[ResizeImage(Width = 100, Height = 100, Mode = ImageResizeMode.Crop)]

public async Task<IActionResult> Upload(IFormFile file)

{

//...

}

```

## JSON

# JSON

The ABP Framework provides an abstraction to work with JSON. Having such an abstraction has some benefits;

\* You can write library independent code. Therefore, you can change the underlying library with the minimum effort and code change.

\* You can use the predefined converters defined in the ABP without worrying about the underlying library's internal details.

> The JSON serialization system is implemented with the [Volo.Abp.Json](https://www.nuget.org/packages/Volo.Abp.Json) NuGet package([Volo.Abp.Json.SystemTextJson](https://www.nuget.org/packages/Volo.Abp.Json.SystemTextJson) is the default implementation). Most of the time, you don't need to manually [install it](https://abp.io/package-detail/Volo.Abp.Json) since it comes pre-installed with the [application startup template](Startup-Templates/Application.md).

## IJsonSerializer

You can inject `IJsonSerializer` and use it for JSON operations. Here is the available operations in the `IJsonSerializer` interface.

```csharp

public interface IJsonSerializer

{

string Serialize(object obj, bool camelCase = true, bool indented = false);

T Deserialize<T>(string jsonString, bool camelCase = true);

object Deserialize(Type type, string jsonString, bool camelCase = true);

}

```

Usage Example:

```csharp

public class ProductManager

{

public IJsonSerializer JsonSerializer { get; }

public ProductManager(IJsonSerializer jsonSerializer)

{

JsonSerializer = jsonSerializer;

}

public void SendRequest(Product product)

{

var json= JsonSerializer.Serialize(product);

// Left blank intentionally for demo purposes...

}

}

```

## Configuration

### AbpJsonOptions

`AbpJsonOptions` type provides options for the JSON operations in the ABP Framework.

Properties:

\* **\*\*InputDateTimeFormats(**`List<string>`**)\*\***: Formats of input JSON date, Empty string means default format. You can provide multiple formats to parse the date.

\* **\*\*OutputDateTimeFormat(**`string`**)\*\***: Format of output json date, Null or empty string means default format.

## System Text Json

### AbpSystemTextJsonSerializerOptions

- **\*\*JsonSerializerOptions(**`System.Text.Json.JsonSerializerOptions`**)\*\***: Global options for System.Text.Json library operations. See [here](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializeroptions) for reference.

### AbpSystemTextJsonSerializerModifiersOptions

- **\*\*Modifiers(**`List<Action<JsonTypeInfo>>`**)\*\***: Configure `Modifiers` of `DefaultJsonTypeInfoResolver`. See [here](https://devblogs.microsoft.com/dotnet/announcing-dotnet-7-preview-6/#json-contract-customization) for reference.

## Newtonsoft

Add [Volo.Abp.Json.Newtonsoft](https://www.nuget.org/packages/Volo.Abp.Json.Newtonsoft) package and depends on `AbpJsonNewtonsoftModule` to replace the `System Text Json`.

#### AbpNewtonsoftJsonSerializerOptions

- **\*\*JsonSerializerSettings(**`Newtonsoft.Json.JsonSerializerSettings`**)\*\***: Global options for Newtonsoft library operations. See [here](https://www.newtonsoft.com/json/help/html/T\_Newtonsoft\_Json\_JsonSerializerSettings.htm) for reference.

## Configuring JSON options in ASP.NET Core

You can change the JSON behavior in ASP.NET Core by configuring [JsonOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.jsonoptions) or

[MvcNewtonsoftJsonOptions](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcnewtonsoftjsonoptions)(if you use `Newtonsoft.Json`)

## Object to Object Mapping

# Object To Object Mapping

It's common to map an object to another similar object. It's also tedious and repetitive since generally both classes have the same or similar properties mapped to each other. Imagine a typical [application service](Application-Services.md) method below:

```csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

}

public async Task CreateUser(CreateUserInput input)

{

//Manually creating a User object from the CreateUserInput object

var user = new User

{

Name = input.Name,

Surname = input.Surname,

EmailAddress = input.EmailAddress,

Password = input.Password

};

await \_userRepository.InsertAsync(user);

}

}

```

`CreateUserInput` is a simple [DTO](Data-Transfer-Objects.md) class and the `User` is a simple [entity](Entities.md). The code above creates a `User` entity from the given input object. The `User` entity will have more properties in a real-world application and manually creating it will become tedious and error-prone. You also have to change the mapping code when you add new properties to `User` and `CreateUserInput` classes.

We can use a library to automatically handle these kind of mappings. ABP provides abstractions for object to object mapping and has an integration package to use [AutoMapper](http://automapper.org/) as the object mapper.

## IObjectMapper

`IObjectMapper` interface (in the [Volo.Abp.ObjectMapping](https://www.nuget.org/packages/Volo.Abp.ObjectMapping) package) defines a simple `Map` method. The example code introduced before can be re-written as shown below:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

}

public async Task CreateUser(CreateUserInput input)

{

//Automatically creating a new User object using the CreateUserInput object

var user = ObjectMapper.Map<CreateUserInput, User>(input);

await \_userRepository.InsertAsync(user);

}

}

````

> `ObjectMapper` is defined in the `ApplicationService` base class in this example. You can directly inject the `IObjectMapper` interface when you need it somewhere else.

Map method has two generic argument: First one is the source object type while the second one is the destination object type.

If you need to set properties of an existing object, you can use the second overload of the `Map` method:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

}

public async Task UpdateUserAsync(Guid id, UpdateUserInput input)

{

var user = await \_userRepository.GetAsync(id);

//Automatically set properties of the user object using the UpdateUserInput

ObjectMapper.Map<UpdateUserInput, User>(input, user);

await \_userRepository.UpdateAsync(user);

}

}

````

You should have defined the mappings before to be able to map objects. See the AutoMapper integration section to learn how to define mappings.

## AutoMapper Integration

[AutoMapper](http://automapper.org/) is one of the most popular object to object mapping libraries. [Volo.Abp.AutoMapper](https://www.nuget.org/packages/Volo.Abp.AutoMapper) package defines the AutoMapper integration for the `IObjectMapper`.

Once you define mappings described as below, you can use the `IObjectMapper` interface just like explained before.

### Define Mappings

AutoMapper provides multiple ways of defining mapping between classes. Refer to [its own documentation](https://docs.automapper.org) for all details.

One way to define object mappings is creating a [Profile](https://docs.automapper.org/en/stable/Configuration.html#profile-instances) class. Example:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<User, UserDto>();

}

}

````

You should then register profiles using the `AbpAutoMapperOptions`:

````csharp

[DependsOn(typeof(AbpAutoMapperModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAutoMapperOptions>(options =>

{

//Add all mappings defined in the assembly of the MyModule class

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](https://docs.automapper.org/en/stable/Attribute-mapping.html).

### Configuration Validation

`AddMaps` optionally takes a `bool` parameter to control the [configuration validation](https://docs.automapper.org/en/stable/Configuration-validation.html) for your [module](Module-Development-Basics.md):

````csharp

options.AddMaps<MyModule>(validate: true);

````

While this option is `false` by default, it is suggested to enable configuration validation as a best practice.

Configuration validation can be controlled per profile class using `AddProfile` instead of `AddMaps`:

````csharp

options.AddProfile<MyProfile>(validate: true);

````

> If you have multiple profiles and need to enable validation only for a few of them, first use `AddMaps` without validation, then use `AddProfile` for each profile you want to validate.

### Mapping the Object Extensions

[Object extension system](Object-Extensions.md) allows to define extra properties for existing classes. ABP Framework provides a mapping definition extension to properly map extra properties of two objects.

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<User, UserDto>()

.MapExtraProperties();

}

}

````

It is suggested to use the `MapExtraProperties()` method if both classes are extensible objects (implement the `IHasExtraProperties` interface). See the [object extension document](Object-Extensions.md) for more.

### Other Useful Extension Methods

There are some more extension methods those can simplify your mapping code.

#### Ignoring Audit Properties

It is common to ignore audit properties when you map an object to another.

Assume that you need to map a `ProductDto` ([DTO](Data-Transfer-Objects.md)) to a `Product` [entity](Entities.md) and the entity is inheriting from the `AuditedEntity` class (which provides properties like `CreationTime`, `CreatorId`, `IHasModificationTime`... etc).

You probably want to ignore these base properties while mapping from the DTO. You can use `IgnoreAuditedObjectProperties()` method to ignore all audit properties (instead of manually ignoring them one by one):

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<ProductDto, Product>()

.IgnoreAuditedObjectProperties();

}

}

````

There are more extension methods like `IgnoreFullAuditedObjectProperties()` and `IgnoreCreationAuditedObjectProperties()` those can be used based on your entity type.

> See the "*\*Base Classes & Interfaces for Audit Properties\**" section in the [entities document](Entities.md) to know more about auditing properties.

#### Ignoring Other Properties

In AutoMapper, you typically write such a mapping code to ignore a property:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<SimpleClass1, SimpleClass2>()

.ForMember(x => x.CreationTime, map => map.Ignore());

}

}

````

We found it unnecessarily long and created the `Ignore()` extension method:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<SimpleClass1, SimpleClass2>()

.Ignore(x => x.CreationTime);

}

}

````

## Advanced Topics

### IObjectMapper<TContext> Interface

Assume that you have created a **\*\*reusable module\*\*** which defines AutoMapper profiles and uses `IObjectMapper` when it needs to map objects. Your module then can be used in different applications, by nature of the [modularity](Module-Development-Basics.md).

`IObjectMapper` is an abstraction and can be replaced by the final application to use another mapping library. The problem here that your reusable module is designed to use the AutoMapper library, because it only defines mappings for it. In such a case, you will want to guarantee that your module always uses AutoMapper even if the final application uses another default object mapping library.

`IObjectMapper<TContext>` is used to contextualize the object mapper, so you can use different libraries for different modules/contexts.

Example usage:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

private readonly IObjectMapper<MyModule> \_objectMapper;

public UserAppService(

IRepository<User, Guid> userRepository,

IObjectMapper<MyModule> objectMapper) //Inject module specific mapper

{

\_userRepository = userRepository;

\_objectMapper = objectMapper;

}

public async Task CreateUserAsync(CreateUserInput input)

{

//Use the module specific mapper

var user = \_objectMapper.Map<CreateUserInput, User>(input);

await \_userRepository.InsertAsync(user);

}

}

````

`UserAppService` injects the `IObjectMapper<MyModule>`, the specific object mapper for this module. It's usage is exactly same of the `IObjectMapper`.

The example code above don't use the `ObjectMapper` property defined in the `ApplicationService`, but injects the `IObjectMapper<MyModule>`. However, it is still possible to use the base property since the `ApplicationService` defines an `ObjectMapperContext` property that can be set in the class constructor. So, the example about can be re-written as like below:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

//Set the object mapper context

ObjectMapperContext = typeof(MyModule);

}

public async Task CreateUserAsync(CreateUserInput input)

{

var user = ObjectMapper.Map<CreateUserInput, User>(input);

await \_userRepository.InsertAsync(user);

}

}

````

While using the contextualized object mapper is same as the normal object mapper, you should register the contextualized mapper in your module's `ConfigureServices` method:

````csharp

[DependsOn(typeof(AbpAutoMapperModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Use AutoMapper for MyModule

context.Services.AddAutoMapperObjectMapper<MyModule>();

Configure<AbpAutoMapperOptions>(options =>

{

options.AddMaps<MyModule>(validate: true);

});

}

}

````

`IObjectMapper<MyModule>` is an essential feature for a reusable module where it can be used in multiple applications each may use a different library for object to object mapping. All pre-built ABP modules are using it. But, for the final application, you can ignore this interface and always use the default `IObjectMapper` interface.

### IObjectMapper<TSource, TDestination> Interface

ABP allows you to customize the mapping code for specific classes. Assume that you want to create a custom class to map from `User` to `UserDto`. In this case, you can create a class that implements the `IObjectMapper<User, UserDto>`:

````csharp

public class MyCustomUserMapper : IObjectMapper<User, UserDto>, ITransientDependency

{

public UserDto Map(User source)

{

//TODO: Create a new UserDto

}

public UserDto Map(User source, UserDto destination)

{

//TODO: Set properties of an existing UserDto

return destination;

}

}

````

ABP automatically discovers and registers the `MyCustomUserMapper` and it is automatically used whenever you use the `IObjectMapper` to map `User` to `UserDto`. A single class may implement more than one `IObjectMapper<TSource, TDestination>` each for a different object pairs.

> This approach is powerful since `MyCustomUserMapper` can inject any other service and use in the `Map` methods.

Once you implement `IObjectMapper<User, UserDto>`, ABP can automatically convert a collection of `User` objects to a collection of `UserDto` objects. The following generic collection types are supported:

\* `IEnumerable<T>`

\* `ICollection<T>`

\* `Collection<T>`

\* `IList<T>`

\* `List<T>`

\* `T[]` (array)

**\*\*Example:\*\***

````csharp

var users = await \_userRepository.GetListAsync(); // returns List<User>

var dtos = ObjectMapper.Map<List<User>, List<UserDto>>(users); // creates List<UserDto>

````

## Simple State Checker

# Simple State Checker

The simple state checking system can be used to enable/disable an object based on some dynamic conditions. For example, you can disable a menu item on the user interface, if the current user has not granted for a given permission. The simple state checking system provides a generic way to define and check such conditions.

## Definition state checker.

Any class can inherit `IHasSimpleStateCheckers` to support state checks.

````csharp

public class MyObject : IHasSimpleStateCheckers<MyObject>

{

public int Id { get; set; }

public List<ISimpleStateChecker<MyObject>> SimpleStateCheckers { get; }

public MyObject()

{

SimpleStateCheckers = new List<ISimpleStateChecker<MyObject>>();

}

}

````

The `MyObject` class contains a collection of state checkers, you can add your custom checkers to it.

````csharp

public class MyObjectStateChecker : ISimpleStateChecker<MyObject>

{

public Task<bool> IsEnabledAsync(SimpleStateCheckerContext<MyObject> context)

{

var currentUser = context.ServiceProvider.GetRequiredService<ICurrentUser>();

return Task.FromResult(currentUser.IsInRole("Admin"));

}

}

````

````csharp

var myobj = new MyObject()

{

Id = 100

};

myobj.SimpleStateCheckers.Add(new MyObjectStateChecker());

````

## Definition Global State Checkers

`AbpSimpleStateCheckerOptions` is the options class that used to set the global state checkers for specific object.

Example: Add global state for `MyObject`:

````csharp

services.Configure<AbpSimpleStateCheckerOptions<MyObject>>(options =>

{

options.GlobalSimpleStateCheckers.Add<MyGlobalObjectStateChecker>();

//options.GlobalSimpleStateCheckers.Add<>(); //Add more global state checkers

});

````

> Write this inside the `ConfigureServices` method of your module.

## Check the state

You can inject `ISimpleStateCheckerManager<MyObject>` service to check state.

````csharp

bool enabled = await stateCheckerManager.IsEnabledAsync(myobj);

````

### Batch check the states

If there are many instance items that require state checking, there may be performance problems.

In this case, you can implement `ISimpleBatchStateChecker`. It can check multiple items at once.

You need to make sure that the same `ISimpleBatchStateChecker` instance is added to the `SimpleStateCheckers` of multiple instances.

> `SimpleBatchStateCheckerBase` inherits the `ISimpleBatchStateChecker` interface and implements the `IsEnabledAsync` method of a single object by default.

````csharp

public class MyObjectBatchStateChecker : SimpleBatchStateCheckerBase<MyObject>

{

public override Task<SimpleStateCheckerResult<MyObject>> IsEnabledAsync(SimpleBatchStateCheckerContext<MyObject> context)

{

var result = new SimpleStateCheckerResult<MyObject>(context.States);

foreach (var myObject in context.States)

{

if (myObject.Id > 100)

{

result[myObject] = true;

}

}

return Task.FromResult(result);

}

}

````

````csharp

var myobj1 = new MyObject()

{

Id = 100

};

var myobj2 = new MyObject()

{

Id = 99

};

var myObjectBatchStateChecker = new MyObjectBatchStateChecker();

myobj1.SimpleStateCheckers.Add(myObjectBatchStateChecker);

myobj2.SimpleStateCheckers.Add(myObjectBatchStateChecker);

SimpleStateCheckerResult<MyObject> stateCheckerResult = await stateCheckerManager.IsEnabledAsync(new []{ myobj1, myobj2 });

````

## Built-in State Checkers

The `PermissionDefinition`, `ApplicationMenuItem` and `ToolbarItem` objects have implemented state checks and have built-in general state checkers, you can directly use their extension methods.

````csharp

RequireAuthenticated();

RequirePermissions(bool requiresAll, params string[] permissions);

RequireFeatures(bool requiresAll, params string[] features);

RequireGlobalFeatures(bool requiresAll, params Type[] globalFeatures);

````

## SMS Sending

# SMS Sending

The ABP Framework provides an abstraction to sending SMS. Having such an abstraction has some benefits;

- You can then **\*\*easily change\*\*** your SMS sender without changing your application code.

- If you want to create **\*\*reusable application modules\*\***, you don't need to make assumption about how the SMS are sent.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

```bash

abp add-package Volo.Abp.Sms

```

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Sms).

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Sms](https://www.nuget.org/packages/Volo.Abp.Sms) NuGet package to your project:

```

Install-Package Volo.Abp.Sms

```

2. Add the `AbpSmsModule` to the dependency list of your module:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpSmsModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

```

## Sending SMS

[Inject](Dependency-Injection.md) the `ISmsSender` into any service and use the `SendAsync` method to send a SMS.

**\*\*Example:\*\***

```csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Sms;

namespace MyProject

{

public class MyService : ITransientDependency

{

private readonly ISmsSender \_smsSender;

public MyService(ISmsSender smsSender)

{

\_smsSender = smsSender;

}

public async Task DoItAsync()

{

await \_smsSender.SendAsync(

"+012345678901", // target phone number

"This is test sms..." // message text

);

}

}

}

```

The given `SendAsync` method in the example is an extension method to send an SMS with primitive parameters. In addition, you can pass an `SmsMessage` object which has the following properties:

- `PhoneNumber` (`string`): Target phone number

- `Text` (`string`): Message text

- `Properties` (`Dictionary<string, string>`): Key-value pairs to pass custom arguments

## NullSmsSender

`NullSmsSender` is a the default implementation of the `ISmsSender`. It writes SMS content to the [standard logger](Logging.md), rather than actually sending the SMS.

This class can be useful especially in development time where you generally don't want to send real SMS. **\*\*However, if you want to actually send SMS, you should implement the** `ISmsSender` **in your application code.\*\***

## Implementing the ISmsSender

You can easily create your SMS sending implementation by creating a class that implements the `ISmsSender` interface, as shown below:

```csharp

using System.IO;

using System.Threading.Tasks;

using Volo.Abp.Sms;

using Volo.Abp.DependencyInjection;

namespace AbpDemo

{

public class MyCustomSmsSender : ISmsSender, ITransientDependency

{

public async Task SendAsync(SmsMessage smsMessage)

{

// Send sms

}

}

}

```

## More

[ABP Commercial](https://commercial.abp.io/) provides Twilio integration package to send SMS over [Twilio service](https://docs.abp.io/en/commercial/latest/modules/twilio-sms).

## String Encryption

# String Encryption

ABP Framework provides string encryption feature that allows to **\*\*Encrypt\*\*** and **\*\*Decrypt\*\*** strings.

## Installation

> This package is already installed by default with the startup template. So, most of the time, you don't need to install it manually.

If installation is needed, it is suggested to use the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

```bash

abp add-package Volo.Abp.Security

```

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Security](https://www.nuget.org/packages/Volo.Abp.Security) NuGet package to your project:

`Install-Package Volo.Abp.Security`

2. Add the `AbpSecurityModule` to the dependency list of your module:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpSecurityModule) // <-- Add module dependency like that

)]

public class YourModule : AbpModule

{

}

```

## Using String Encryption

All encryption operations are included in `IStringEncryptionService`. You can inject it and start to use.

```csharp

public class MyService : DomainService

{

protected IStringEncryptionService StringEncryptionService { get; }

public MyService(IStringEncryptionService stringEncryptionService)

{

StringEncryptionService = stringEncryptionService;

}

public string Encrypt(string value)

{

// To enrcypt a value

return StringEncryptionService.Encrypt(value);

}

public string Decrpyt(string value)

{

// To decrypt a value

return StringEncryptionService.Decrypt(value);

}

}

```

### Using Custom PassPhrase

`IStringEncryptionService` methods has **\*\*passPharase\*\*** parameter with default value and it uses default PassPhrase when you don't pass passPhrase parameter.

```csharp

// Default Pass Phrase

var encryptedValue = StringEncryptionService.Encrypt(value);

// Custom Pass Phrase

var encryptedValue = StringEncryptionService.Encrypt(value, "MyCustomPassPhrase");

// Encrypt & Decrypt have same parameters.

var decryptedValue = StringEncryptionService.Decrypt(value, "MyCustomPassPhrase");

```

### Using Custom Salt

`IStringEncryptionService` methods has **\*\*salt\*\*** parameter with default value and it uses default Salt when you don't pass the parameter.

```csharp

// Default Salt

var encryptedValue = StringEncryptionService.Encrypt(value);

// Custom Salt

var encryptedValue = StringEncryptionService.Encrypt(value, salt: Encoding.UTF8.GetBytes("MyCustomSalt"));

// Encrypt & Decrypt have same parameters.

var decryptedValue = StringEncryptionService.Decrypt(value, salt: Encoding.UTF8.GetBytes("MyCustomSalt"));

```

\*\*\*

## String Encryption Options

Default values can be configured with `AbpStringEncryptionOptions` type.

```csharp

Configure<AbpStringEncryptionOptions>(opts =>

{

opts.DefaultPassPhrase = "MyStrongPassPhrase";

opts.DefaultSalt = Encoding.UTF8.GetBytes("MyStrongSalt");

opts.InitVectorBytes = Encoding.UTF8.GetBytes("YetAnotherStrongSalt");

opts.Keysize = 512;

});

```

- **\*\*DefaultPassPhrase\*\***: Default password to encrypt/decrypt texts. It's recommended to set to another value for security. Default value: `gsKnGZ041HLL4IM8`

- **\*\*DefaultSalt\*\***: A value which is used as salt while encrypting/decrypting.

Default value: `Encoding.ASCII.GetBytes("hgt!16kl")`

- **\*\*InitVectorBytes:\*\*** This constant string is used as a "salt" value for the PasswordDeriveBytes function calls. This size of the IV (in bytes) must = (keysize / 8). Default keysize is 256, so the IV must be 32 bytes long. Using a 16 character string here gives us 32 bytes when converted to a byte array.

Default value: `Encoding.ASCII.GetBytes("jkE49230Tf093b42")`

- **\*\*Keysize:\*\*** This constant is used to determine the keysize of the encryption algorithm.

Default value: `256`

## Text Templating

# Text Templating

## Introduction

ABP Framework provides a simple, yet efficient text template system. Text templating is used to dynamically render contents based on a template and a model (a data object):

Template + Model =renderer=> Rendered Content

It is very similar to an ASP.NET Core Razor View (or Page):

*\*RAZOR VIEW (or PAGE) + MODEL* ***==render==****> HTML CONTENT\**

You can use the rendered output for any purpose, like sending emails or preparing some reports.

Template rendering engine is very powerful;

\* It supports **\*\*conditional logics\*\***, **\*\*loops\*\*** and much more.

\* Template content **\*\*can be localized\*\***.

\* You can define **\*\*layout templates\*\*** to be used as the layout while rendering other templates.

\* You can pass arbitrary objects to the template context (beside the model) for advanced scenarios.

ABP Framework provides two templating engines;

\* **\*\*[**Razor**](Text-Templating-Razor.md)\*\***

\* **\*\*[**Scriban**](Text-Templating-Scriban.md)\*\***

You can use different template engines in the same application, or even create a new custom template engine.

## Source Code

Get [the source code of the sample application](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo) developed and referred through this document.

## See Also

\* [The source code of the sample application](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo) developed and referred through this document.

\* [Localization system](Localization.md).

\* [Virtual File System](Virtual-File-System.md).

### Razor Integration

# Razor Integration

The Razor template is a standard C# class, so you can freely use the functions of C#, such as `dependency injection`, using `LINQ`, custom methods, and even using `Repository`.

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.TextTemplating.Razor

````

> If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.TextTemplating.Razor).

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.TextTemplating.Razor](https://www.nuget.org/packages/Volo.Abp.TextTemplating.Razor) NuGet package to your project:

````

Install-Package Volo.Abp.TextTemplating.Razor

````

2. Add the `AbpTextTemplatingRazorModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpTextTemplatingRazorModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## Add MetadataReference to CSharpCompilerOptions

You need to add the `MetadataReference` of the type used in the template to `CSharpCompilerOptions's References`.

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpRazorTemplateCSharpCompilerOptions>(options =>

{

options.References.Add(MetadataReference.CreateFromFile(typeof(YourModule).Assembly.Location));

});

}

````

## Add MetadataReference for a template.

You can add some `MetadataReference` to the template

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

services.Configure<AbpCompiledViewProviderOptions>(options =>

{

//Hello is template name.

options.TemplateReferences.Add("Hello", new List<Assembly>()

{

Assembly.Load("Microsoft.Extensions.Logging.Abstractions"),

Assembly.Load("Microsoft.Extensions.Logging")

}

.Select(x => MetadataReference.CreateFromFile(x.Location))

.ToList());

});

}

````

## Defining Templates

Before rendering a template, you should define it. Create a class inheriting from the `TemplateDefinitionProvider` base class:

````csharp

public class DemoTemplateDefinitionProvider : TemplateDefinitionProvider

{

public override void Define(ITemplateDefinitionContext context)

{

context.Add(

new TemplateDefinition("Hello") //template name: "Hello"

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/Hello/Hello.cshtml", //template content path

isInlineLocalized: true

)

);

}

}

````

\* `context` object is used to add new templates or get the templates defined by depended modules. Used `context.Add(...)` to define a new template.

\* `TemplateDefinition` is the class represents a template. Each template must have a unique name (that will be used while you are rendering the template).

\* `/Demos/Hello/Hello.cshtml` is the path of the template file.

\* `isInlineLocalized` is used to declare if you are using a single template for all languages (`true`) or different templates for each language (`false`). See the Localization section below for more.

\* `WithRenderEngine` method is used to set the render engine of the template.

### The Template Base

Every `cshtml` template page needs to inherit `RazorTemplatePageBase` or `RazorTemplatePageBase<Model>`.

There are some useful properties in the base class that can be used in templates. eg: `Localizer`, `ServiceProvider`.

### The Template Content

`WithVirtualFilePath` indicates that we are using the [Virtual File System](Virtual-File-System.md) to store the template content. Create a `Hello.cshtml` file inside your project and mark it as "**\*\*embedded resource\*\***" on the properties window:

![hello-template-razor](images/hello-template-razor.png)

Example `Hello.cshtml` content is shown below:

````

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase<HelloModelNamespace.HelloModel>

Hello @Model.Name

````

The `HelloModel` class is:

````csharp

namespace HelloModelNamespace

{

public class HelloModel

{

public string Name { get; set; }

}

}

````

The [Virtual File System](Virtual-File-System.md) requires to add your files in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class:

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<TextTemplateDemoModule>("TextTemplateDemo");

});

````

\* `TextTemplateDemoModule` is the module class that you define your template in.

\* `TextTemplateDemo` is the root namespace of your project.

## Rendering the Template

`ITemplateRenderer` service is used to render a template content.

### Example: Rendering a Simple Template

````csharp

public class HelloDemo : ITransientDependency

{

private readonly ITemplateRenderer \_templateRenderer;

public HelloDemo(ITemplateRenderer templateRenderer)

{

\_templateRenderer = templateRenderer;

}

public async Task RunAsync()

{

var result = await \_templateRenderer.RenderAsync(

"Hello", //the template name

new HelloModel

{

Name = "John"

}

);

Console.WriteLine(result);

}

}

````

\* `HelloDemo` is a simple class that injects the `ITemplateRenderer` in its constructor and uses it inside the `RunAsync` method.

\* `RenderAsync` gets two fundamental parameters:

\* `templateName`: The name of the template to be rendered (`Hello` in this example).

\* `model`: An object that is used as the `model` inside the template (a `HelloModel` object in this example).

The result shown below for this example:

````csharp

Hello John :)

````

## Localization

It is possible to localize a template content based on the current culture. There are two types of localization options described in the following sections.

### Inline localization

Inline localization uses the [localization system](Localization.md) to localize texts inside templates.

#### Example: Reset Password Link

Assuming you need to send an email to a user to reset her/his password. Here, the model/template content:

````csharp

namespace ResetMyPasswordModelNamespace

{

public class ResetMyPasswordModel

{

public string Link { get; set; }

public string Name { get; set; }

}

}

````

````html

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase<ResetMyPasswordModelNamespace.ResetMyPasswordModel>

<a title="@Localizer["ResetMyPasswordTitle"]" href="@Model.Link">@Localizer["ResetMyPassword", Model.Name]</a>

````

`Localizer` service is used to localize the given key based on the current user culture. You need to define the `ResetMyPassword` key inside your localization file:

````json

"ResetMyPasswordTitle": "Reset my password",

"ResetMyPassword": "Hi {0}, Click here to reset your password"

````

You also need to declare the localization resource to be used with this template, inside your template definition provider class:

````csharp

context.Add(

new TemplateDefinition(

"PasswordReset", //Template name

typeof(DemoResource) //LOCALIZATION RESOURCE

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/PasswordReset/PasswordReset.cshtml", //template content path

isInlineLocalized: true

)

);

````

That's all. When you render this template like that:

````csharp

var result = await \_templateRenderer.RenderAsync(

"PasswordReset", //the template name

new PasswordResetModel

{

Name = "john",

Link = "https://abp.io/example-link?userId=123&token=ABC"

}

);

````

You will see the localized result:

````html

<a title="Reset my password" href="https://abp.io/example-link?userId=123&token=ABC">Hi john, Click here to reset your password</a>

````

> If you define the [default localization resource](Localization.md) for your application, then no need to declare the resource type for the template definition.

### Multiple Contents Localization

Instead of a single template that uses the localization system to localize the template, you may want to create different template files for each language. It can be needed if the template should be completely different for a specific culture rather than simple text localizations.

#### Example: Welcome Email Template

Assuming that you want to send a welcome email to your users, but want to define a completely different template based on the user culture.

First, create a folder and put your templates inside it, like `en.cshtml`, `tr.cshtml`... one for each culture you support:

![multiple-file-template-razor](images/multiple-file-template-razor.png)

Then add your template definition in the template definition provider class:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en"

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates", //template content folder

isInlineLocalized: false

)

);

````

\* Set **\*\*default culture name\*\***, so it fallbacks to the default culture if there is no template for the desired culture.

\* Specify **\*\*the template folder\*\*** rather than a single template file.

\* Set `isInlineLocalized` to `false` for this case.

That's all, you can render the template for the current culture:

````csharp

var result = await \_templateRenderer.RenderAsync("WelcomeEmail");

````

> Skipped the modal for this example to keep it simple, but you can use models as just explained before.

### Specify the Culture

`ITemplateRenderer` service uses the current culture (`CultureInfo.CurrentUICulture`) if not specified. If you need, you can specify the culture as the `cultureName` parameter:

````csharp

var result = await \_templateRenderer.RenderAsync(

"WelcomeEmail",

cultureName: "en"

);

````

## Layout Templates

Layout templates are used to create shared layouts among other templates. It is similar to the layout system in the ASP.NET Core MVC / Razor Pages.

### Example: Email HTML Layout Template

For example, you may want to create a single layout for all of your email templates.

First, create a template file just like before:

````html

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

@Body

</body>

</html>

````

\* A layout template must have a `Body` part as a place holder for the rendered child content.

The register your template in the template definition provider:

````csharp

context.Add(

new TemplateDefinition(

"EmailLayout",

isLayout: true //SET isLayout!

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/EmailLayout/EmailLayout.cshtml",

isInlineLocalized: true

)

);

````

Now, you can use this template as the layout of any other template:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en",

layout: "EmailLayout" //Set the LAYOUT

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates",

isInlineLocalized: false

)

);

````

## Global Context

ABP passes the `model` that can be used to access to the model inside the template. You can pass more global variables if you need.

An example template content:

````html

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase

A global object value: @GlobalContext["myGlobalObject"]

````

This template assumes that that is a `myGlobalObject` object in the template rendering context. You can provide it like shown below:

````csharp

var result = await \_templateRenderer.RenderAsync(

"GlobalContextUsage",

globalContext: new Dictionary<string, object>

{

{"myGlobalObject", "TEST VALUE"}

}

);

````

The rendering result will be:

````

A global object value: TEST VALUE

````

## Replacing the Existing Templates

It is possible to replace a template defined by a module that used in your application. In this way, you can customize the templates based on your requirements without changing the module code.

### Option-1: Using the Virtual File System

The [Virtual File System](Virtual-File-System.md) allows you to override any file by placing the same file into the same path in your project.

#### Example: Replace the Standard Email Layout Template

ABP Framework provides an [email sending system](Emailing.md) that internally uses the text templating to render the email content. It defines a standard email layout template in the `/Volo/Abp/Emailing/Templates/Layout.cshtml` path. The unique name of the template is `Abp.StandardEmailTemplates.Layout` and this string is defined as a constant on the `Volo.Abp.Emailing.Templates.StandardEmailTemplates` static class.

Do the following steps to replace the template file with your own;

**\*\*1)\*\*** Add a new file into the same location (`/Volo/Abp/Emailing/Templates/Layout.cshtml`) in your project:

![replace-email-layout-razor](images/replace-email-layout-razor.png)

**\*\*2)\*\*** Prepare your email layout template:

````html

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

<h1>This my header</h1>

@Body

<footer>

This is my footer...

</footer>

</body>

</html>

````

This example simply adds a header and footer to the template and renders the content between them (see the *\*Layout Templates\** section above to understand it).

**\*\*3)\*\*** Configure the embedded resources in the `.csproj` file

\* Add [Microsoft.Extensions.FileProviders.Embedded](https://www.nuget.org/packages/Microsoft.Extensions.FileProviders.Embedded) NuGet package to the project.

\* Add `<GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>` into the `<PropertyGroup>...</PropertyGroup>` section of your `.csproj` file.

\* Add the following code into your `.csproj` file:

````xml

<ItemGroup>

<None Remove="Volo\Abp\Emailing\Templates\\*.cshtml" />

<EmbeddedResource Include="Volo\Abp\Emailing\Templates\\*.cshtml" />

</ItemGroup>

````

This makes the template files "embedded resource".

**\*\*4)\*\*** Configure the virtual file system

Configure the `AbpVirtualFileSystemOptions` in the `ConfigureServices` method of your [module](Module-Development-Basics.md) to add the embedded files into the virtual file system:

```csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<BookStoreDomainModule>();

});

```

`BookStoreDomainModule` should be your module name, in this example code.

> Be sure that your module (directly or indirectly) [depends on](Module-Development-Basics.md) the `AbpEmailingModule`. Because the VFS can override files based on the dependency order.

Now, your template will be used when you want to render the email layout template.

### Option-2: Using the Template Definition Provider

You can create a template definition provider class that gets the email layout template and changes the virtual file path for the template.

**\*\*Example: Use the** `/MyTemplates/EmailLayout.cshtml` **file instead of the standard template\*\***

```csharp

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing.Templates;

using Volo.Abp.TextTemplating;

namespace MyProject

{

public class MyTemplateDefinitionProvider

: TemplateDefinitionProvider, ITransientDependency

{

public override void Define(ITemplateDefinitionContext context)

{

var emailLayoutTemplate = context.GetOrNull(StandardEmailTemplates.Layout);

emailLayoutTemplate

.WithVirtualFilePath(

"/MyTemplates/EmailLayout.cshtml",

isInlineLocalized: true

);

}

}

}

```

You should still add the file `/MyTemplates/EmailLayout.cshtml` to the virtual file system as explained before. This approach allows you to locate templates in any folder instead of the folder defined by the depended module.

Beside the template content, you can manipulate the template definition properties, like `DisplayName`, `Layout` or `LocalizationSource`.

## Advanced Features

This section covers some internals and more advanced usages of the text templating system.

### Template Content Provider

`ITemplateRenderer` is used to render the template, which is what you want for most of the cases. However, you can use the `ITemplateContentProvider` to get the raw (not rendered) template contents.

> `ITemplateContentProvider` is internally used by the `ITemplateRenderer` to get the raw template contents.

Example:

````csharp

public class TemplateContentDemo : ITransientDependency

{

private readonly ITemplateContentProvider \_templateContentProvider;

public TemplateContentDemo(ITemplateContentProvider templateContentProvider)

{

\_templateContentProvider = templateContentProvider;

}

public async Task RunAsync()

{

var result = await \_templateContentProvider

.GetContentOrNullAsync("Hello");

Console.WriteLine(result);

}

}

````

The result will be the raw template content:

````

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase<HelloModelNamespace.HelloModel>

Hello @Model.Name

````

\* `GetContentOrNullAsync` returns `null` if no content defined for the requested template.

\* It can get a `cultureName` parameter that is used if template has different files for different cultures (see Multiple Contents Localization section above).

### Template Content Contributor

`ITemplateContentProvider` service uses `ITemplateContentContributor` implementations to find template contents. There is a single pre-implemented content contributor, `VirtualFileTemplateContentContributor`, which gets template contents from the virtual file system as described above.

You can implement the `ITemplateContentContributor` to read raw template contents from another source.

Example:

````csharp

public class MyTemplateContentProvider

: ITemplateContentContributor, ITransientDependency

{

public async Task<string> GetOrNullAsync(TemplateContentContributorContext context)

{

var templateName = context.TemplateDefinition.Name;

//TODO: Try to find content from another source

return null;

}

}

````

Return `null` if your source can not find the content, so `ITemplateContentProvider` fallbacks to the next contributor.

### Template Definition Manager

`ITemplateDefinitionManager` service can be used to get the template definitions (created by the template definition providers).

## See Also

\* [The source code of the sample application](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo) developed and referred through this document.

\* [Localization system](Localization.md).

\* [Virtual File System](Virtual-File-System.md).

### Scriban Integration

# Scriban Integration

## Installation

It is suggested to use the [ABP CLI](CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the project (.csproj file) and type the following command:

````bash

abp add-package Volo.Abp.TextTemplating.Scriban

````

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.TextTemplating.Scriban](https://www.nuget.org/packages/Volo.Abp.TextTemplating.Scriban) NuGet package to your project:

````

Install-Package Volo.Abp.TextTemplating.Scriban

````

2. Add the `AbpTextTemplatingScribanModule` to the dependency list of your module:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpTextTemplatingScribanModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## Defining Templates

Before rendering a template, you should define it. Create a class inheriting from the `TemplateDefinitionProvider` base class:

````csharp

public class DemoTemplateDefinitionProvider : TemplateDefinitionProvider

{

public override void Define(ITemplateDefinitionContext context)

{

context.Add(

new TemplateDefinition("Hello") //template name: "Hello"

.WithVirtualFilePath(

"/Demos/Hello/Hello.tpl", //template content path

isInlineLocalized: true

)

.WithScribanEngine()

);

}

}

````

\* `context` object is used to add new templates or get the templates defined by depended modules. Used `context.Add(...)` to define a new template.

\* `TemplateDefinition` is the class represents a template. Each template must have a unique name (that will be used while you are rendering the template).

\* `/Demos/Hello/Hello.tpl` is the path of the template file.

\* `isInlineLocalized` is used to declare if you are using a single template for all languages (`true`) or different templates for each language (`false`). See the Localization section below for more.

\* `WithRenderEngine` method is used to set the render engine of the template.

### The Template Content

`WithVirtualFilePath` indicates that we are using the [Virtual File System](Virtual-File-System.md) to store the template content. Create a `Hello.tpl` file inside your project and mark it as "**\*\*embedded resource\*\***" on the properties window:

![hello-template](images/hello-template.png)

Example `Hello.tpl` content is shown below:

````

Hello {%{{{model.name}}}%} :)

````

The [Virtual File System](Virtual-File-System.md) requires to add your files in the `ConfigureServices` method of your [module](Module-Development-Basics.md) class:

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<TextTemplateDemoModule>("TextTemplateDemo");

});

````

\* `TextTemplateDemoModule` is the module class that you define your template in.

\* `TextTemplateDemo` is the root namespace of your project.

## Rendering the Template

`ITemplateRenderer` service is used to render a template content.

### Example: Rendering a Simple Template

````csharp

public class HelloDemo : ITransientDependency

{

private readonly ITemplateRenderer \_templateRenderer;

public HelloDemo(ITemplateRenderer templateRenderer)

{

\_templateRenderer = templateRenderer;

}

public async Task RunAsync()

{

var result = await \_templateRenderer.RenderAsync(

"Hello", //the template name

new HelloModel

{

Name = "John"

}

);

Console.WriteLine(result);

}

}

````

\* `HelloDemo` is a simple class that injects the `ITemplateRenderer` in its constructor and uses it inside the `RunAsync` method.

\* `RenderAsync` gets two fundamental parameters:

\* `templateName`: The name of the template to be rendered (`Hello` in this example).

\* `model`: An object that is used as the `model` inside the template (a `HelloModel` object in this example).

The result shown below for this example:

````csharp

Hello John :)

````

### Anonymous Model

While it is suggested to create model classes for the templates, it would be practical (and possible) to use anonymous objects for simple cases:

````csharp

var result = await \_templateRenderer.RenderAsync(

"Hello",

new

{

Name = "John"

}

);

````

In this case, we haven't created a model class, but created an anonymous object as the model.

### PascalCase vs snake\_case

PascalCase property names (like `UserName`) is used as snake\_case (like `user\_name`) in the templates.

## Localization

It is possible to localize a template content based on the current culture. There are two types of localization options described in the following sections.

### Inline localization

Inline localization uses the [localization system](Localization.md) to localize texts inside templates.

#### Example: Reset Password Link

Assuming you need to send an email to a user to reset her/his password. Here, the template content:

````

<a title="{%{{{L "ResetMyPasswordTitle"}}}%}" href="{%{{{model.link}}}%}">{%{{{L "ResetMyPassword" model.name}}}%}</a>

````

`L` function is used to localize the given key based on the current user culture. You need to define the `ResetMyPassword` key inside your localization file:

````json

"ResetMyPasswordTitle": "Reset my password",

"ResetMyPassword": "Hi {0}, Click here to reset your password"

````

You also need to declare the localization resource to be used with this template, inside your template definition provider class:

````csharp

context.Add(

new TemplateDefinition(

"PasswordReset", //Template name

typeof(DemoResource) //LOCALIZATION RESOURCE

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/PasswordReset/PasswordReset.tpl", //template content path

isInlineLocalized: true

)

);

````

That's all. When you render this template like that:

````csharp

var result = await \_templateRenderer.RenderAsync(

"PasswordReset", //the template name

new PasswordResetModel

{

Name = "john",

Link = "https://abp.io/example-link?userId=123&token=ABC"

}

);

````

You will see the localized result:

````csharp

<a title="Reset my password" href="https://abp.io/example-link?userId=123&token=ABC">Hi john, Click here to reset your password</a>

````

> If you define the [default localization resource](Localization.md) for your application, then no need to declare the resource type for the template definition.

### Multiple Contents Localization

Instead of a single template that uses the localization system to localize the template, you may want to create different template files for each language. It can be needed if the template should be completely different for a specific culture rather than simple text localizations.

#### Example: Welcome Email Template

Assuming that you want to send a welcome email to your users, but want to define a completely different template based on the user culture.

First, create a folder and put your templates inside it, like `en.tpl`, `tr.tpl`... one for each culture you support:

![multiple-file-template](images/multiple-file-template.png)

Then add your template definition in the template definition provider class:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en"

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates", //template content folder

isInlineLocalized: false

)

);

````

\* Set **\*\*default culture name\*\***, so it fallbacks to the default culture if there is no template for the desired culture.

\* Specify **\*\*the template folder\*\*** rather than a single template file.

\* Set `isInlineLocalized` to `false` for this case.

That's all, you can render the template for the current culture:

````csharp

var result = await \_templateRenderer.RenderAsync("WelcomeEmail");

````

> Skipped the modal for this example to keep it simple, but you can use models as just explained before.

### Specify the Culture

`ITemplateRenderer` service uses the current culture (`CultureInfo.CurrentUICulture`) if not specified. If you need, you can specify the culture as the `cultureName` parameter:

````csharp

var result = await \_templateRenderer.RenderAsync(

"WelcomeEmail",

cultureName: "en"

);

````

## Layout Templates

Layout templates are used to create shared layouts among other templates. It is similar to the layout system in the ASP.NET Core MVC / Razor Pages.

### Example: Email HTML Layout Template

For example, you may want to create a single layout for all of your email templates.

First, create a template file just like before:

````xml

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

{%{{{content}}}%}

</body>

</html>

````

\* A layout template must have a **\*\*{%{{{content}}}%}\*\*** part as a place holder for the rendered child content.

The register your template in the template definition provider:

````csharp

context.Add(

new TemplateDefinition(

"EmailLayout",

isLayout: true //SET isLayout!

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/EmailLayout/EmailLayout.tpl",

isInlineLocalized: true

)

);

````

Now, you can use this template as the layout of any other template:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en",

layout: "EmailLayout" //Set the LAYOUT

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates",

isInlineLocalized: false

)

);

````

## Global Context

ABP passes the `model` that can be used to access to the model inside the template. You can pass more global variables if you need.

An example template content:

````

A global object value: {%{{{myGlobalObject}}}%}

````

This template assumes that that is a `myGlobalObject` object in the template rendering context. You can provide it like shown below:

````csharp

var result = await \_templateRenderer.RenderAsync(

"GlobalContextUsage",

globalContext: new Dictionary<string, object>

{

{"myGlobalObject", "TEST VALUE"}

}

);

````

The rendering result will be:

````

A global object value: TEST VALUE

````

## Replacing the Existing Templates

It is possible to replace a template defined by a module that used in your application. In this way, you can customize the templates based on your requirements without changing the module code.

### Option-1: Using the Virtual File System

The [Virtual File System](Virtual-File-System.md) allows you to override any file by placing the same file into the same path in your project.

#### Example: Replace the Standard Email Layout Template

ABP Framework provides an [email sending system](Emailing.md) that internally uses the text templating to render the email content. It defines a standard email layout template in the `/Volo/Abp/Emailing/Templates/Layout.tpl` path. The unique name of the template is `Abp.StandardEmailTemplates.Layout` and this string is defined as a constant on the `Volo.Abp.Emailing.Templates.StandardEmailTemplates` static class.

Do the following steps to replace the template file with your own;

**\*\*1)\*\*** Add a new file into the same location (`/Volo/Abp/Emailing/Templates/Layout.tpl`) in your project:

![replace-email-layout](images/replace-email-layout.png)

**\*\*2)\*\*** Prepare your email layout template:

````html

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

<h1>This my header</h1>

{%{{{content}}}%}

<footer>

This is my footer...

</footer>

</body>

</html>

````

This example simply adds a header and footer to the template and renders the content between them (see the *\*Layout Templates\** section above to understand it).

**\*\*3)\*\*** Configure the embedded resources in the `.csproj` file

\* Add [Microsoft.Extensions.FileProviders.Embedded](https://www.nuget.org/packages/Microsoft.Extensions.FileProviders.Embedded) NuGet package to the project.

\* Add `<GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>` into the `<PropertyGroup>...</PropertyGroup>` section of your `.csproj` file.

\* Add the following code into your `.csproj` file:

````xml

<ItemGroup>

<None Remove="Volo\Abp\Emailing\Templates\\*.tpl" />

<EmbeddedResource Include="Volo\Abp\Emailing\Templates\\*.tpl" />

</ItemGroup>

````

This makes the template files "embedded resource".

**\*\*4)\*\*** Configure the virtual file system

Configure the `AbpVirtualFileSystemOptions` in the `ConfigureServices` method of your [module](Module-Development-Basics.md) to add the embedded files into the virtual file system:

```csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<BookStoreDomainModule>();

});

```

`BookStoreDomainModule` should be your module name, in this example code.

> Be sure that your module (directly or indirectly) [depends on](Module-Development-Basics.md) the `AbpEmailingModule`. Because the VFS can override files based on the dependency order.

Now, your template will be used when you want to render the email layout template.

### Option-2: Using the Template Definition Provider

You can create a template definition provider class that gets the email layout template and changes the virtual file path for the template.

**\*\*Example: Use the** `/MyTemplates/EmailLayout.tpl` **file instead of the standard template\*\***

```csharp

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing.Templates;

using Volo.Abp.TextTemplating;

namespace MyProject

{

public class MyTemplateDefinitionProvider

: TemplateDefinitionProvider, ITransientDependency

{

public override void Define(ITemplateDefinitionContext context)

{

var emailLayoutTemplate = context.GetOrNull(StandardEmailTemplates.Layout);

emailLayoutTemplate

.WithVirtualFilePath(

"/MyTemplates/EmailLayout.tpl",

isInlineLocalized: true

);

}

}

}

```

You should still add the file `/MyTemplates/EmailLayout.tpl` to the virtual file system as explained before. This approach allows you to locate templates in any folder instead of the folder defined by the depended module.

Beside the template content, you can manipulate the template definition properties, like `DisplayName`, `Layout` or `LocalizationSource`.

## Advanced Features

This section covers some internals and more advanced usages of the text templating system.

### Template Content Provider

`ITemplateRenderer` is used to render the template, which is what you want for most of the cases. However, you can use the `ITemplateContentProvider` to get the raw (not rendered) template contents.

> `ITemplateContentProvider` is internally used by the `ITemplateRenderer` to get the raw template contents.

Example:

````csharp

public class TemplateContentDemo : ITransientDependency

{

private readonly ITemplateContentProvider \_templateContentProvider;

public TemplateContentDemo(ITemplateContentProvider templateContentProvider)

{

\_templateContentProvider = templateContentProvider;

}

public async Task RunAsync()

{

var result = await \_templateContentProvider

.GetContentOrNullAsync("Hello");

Console.WriteLine(result);

}

}

````

The result will be the raw template content:

````

Hello {%{{{model.name}}}%} :)

````

\* `GetContentOrNullAsync` returns `null` if no content defined for the requested template.

\* It can get a `cultureName` parameter that is used if template has different files for different cultures (see Multiple Contents Localization section above).

### Template Content Contributor

`ITemplateContentProvider` service uses `ITemplateContentContributor` implementations to find template contents. There is a single pre-implemented content contributor, `VirtualFileTemplateContentContributor`, which gets template contents from the virtual file system as described above.

You can implement the `ITemplateContentContributor` to read raw template contents from another source.

Example:

````csharp

public class MyTemplateContentProvider

: ITemplateContentContributor, ITransientDependency

{

public async Task<string> GetOrNullAsync(TemplateContentContributorContext context)

{

var templateName = context.TemplateDefinition.Name;

//TODO: Try to find content from another source

return null;

}

}

````

Return `null` if your source can not find the content, so `ITemplateContentProvider` fallbacks to the next contributor.

### Template Definition Manager

`ITemplateDefinitionManager` service can be used to get the template definitions (created by the template definition providers).

## See Also

\* [The source code of the sample application](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo) developed and referred through this document.

\* [Localization system](Localization.md).

\* [Virtual File System](Virtual-File-System.md).

## Timing

# Timing

Working with times & [time zones](https://en.wikipedia.org/wiki/Time\_zone) is always tricky, especially if you need to build a **\*\*global system\*\*** that is used by users in **\*\*different time zones\*\***.

ABP provides a basic infrastructure to make it easy and handle automatically wherever possible. This document covers the ABP Framework services and systems related to time and time zones.

> If you are creating a local application that runs in a single time zone region, you may not need all these systems. But even in this case, it is suggested to use the `IClock` service introduced in this document.

## IClock

`DateTime.Now` returns a `DateTime` object with the **\*\*local date & time of the server\*\***. A `DateTime` object **\*\*doesn't store the time zone information\*\***. So, you can not know the **\*\*absolute date & time\*\*** stored in this object. You can only make **\*\*assumptions\*\***, like assuming that it was created in UTC+05 time zone. The things especially gets complicated when you save this value to a database and read later, or send it to a client in a **\*\*different time zone\*\***.

One solution to this problem is always use `DateTime.UtcNow` and assume all `DateTime` objects as UTC time. In this way, you can convert it to the time zone of the target client when needed.

`IClock` provides an abstraction while getting the current time, so you can control the kind of the date time (UTC or local) in a single point in your application.

**\*\*Example: Getting the current time\*\***

````csharp

using Volo.Abp.DependencyInjection;

using Volo.Abp.Timing;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IClock \_clock;

public MyService(IClock clock)

{

\_clock = clock;

}

public void Foo()

{

//Get the current time!

var now = \_clock.Now;

}

}

}

````

\* Inject the `IClock` service when you need to get the current time. Common base classes (like ApplicationService) already injects it and provides as a base property - so, you can directly use as `Clock`.

\* Use the `Now` property to get the current time.

> Most of the times, `IClock` is the only service you need to know and use in your application.

### Clock Options

`AbpClockOptions` is the [options](Options.md) class that used to set the clock kind.

**\*\*Example: Use UTC Clock\*\***

````csharp

Configure<AbpClockOptions>(options =>

{

options.Kind = DateTimeKind.Utc;

});

````

Write this inside the `ConfigureServices` method of your [module](Module-Development-Basics.md).

> Default `Kind` is `Unspecified`, that actually make the Clock as it doesn't exists at all. Either make it `Utc` or `Local` if you want to get benefit of the Clock system.

### DateTime Normalization

Other important function of the `IClock` is to normalize `DateTime` objects.

**\*\*Example usage:\*\***

````csharp

DateTime dateTime = ...; //Get from somewhere

var normalizedDateTime = Clock.Normalize(dateTime)

````

`Normalize` method works as described below:

\* Converts the given `DateTime` to the UTC (by using the `DateTime.ToUniversalTime()` method) if current Clock is UTC and given `DateTime` is local.

\* Converts the given `DateTime` to the local (by using the `DateTime.ToLocalTime()` method) if current Clock is local and given `DateTime` is UTC.

\* Sets `Kind` of the given `DateTime` (using the `DateTime.SpecifyKind(...)` method) to the `Kind` of the current Clock if given `DateTime`'s `Kind` is `Unspecified`.

`Normalize` method is used by the ABP Framework when the it gets a `DateTime` that is not created by `IClock.Now` and may not be compatible with the current Clock type. Examples;

\* `DateTime` type binding in the ASP.NET Core MVC model binding.

\* Saving data to and reading data from database via [Entity Framework Core](Entity-Framework-Core.md).

\* Working with `DateTime` objects on [JSON deserialization](Json-Serialization.md).

#### DisableDateTimeNormalization Attribute

`DisableDateTimeNormalization` attribute can be used to disable the normalization operation for desired classes or properties.

### Other IClock Properties

In addition to the `Now`, `IClock` service has the following properties:

\* `Kind`: Returns a `DateTimeKind` for the currently used clock type (`DateTimeKind.Utc`, `DateTimeKind.Local` or `DateTimeKind.Unspecified`).

\* `SupportsMultipleTimezone`: Returns `true` if currently used clock is UTC.

## Time Zones

This section covers the ABP Framework infrastructure related to managing time zones.

### TimeZone Setting

ABP Framework defines **\*\*a setting\*\***, named `Abp.Timing.Timezone`, that can be used to set and get the time zone for a user, [tenant](Multi-Tenancy.md) or globally for the application. The default value is `UTC`.

See the [setting documentation](Settings.md) to learn more about the setting system.

### ITimezoneProvider

`ITimezoneProvider` is a service to simple convert [Windows Time Zone Id](https://support.microsoft.com/en-us/help/973627/microsoft-time-zone-index-values) values to [Iana Time Zone Name](https://www.iana.org/time-zones) values and vice verse. It also provides methods to get list of these time zones and get a `TimeZoneInfo` with a given name.

It has been implemented using the [TimeZoneConverter](https://github.com/mj1856/TimeZoneConverter) library.

## Virtual File System

# Virtual File System

The Virtual File System makes it possible to manage files that do not physically exist on the file system (disk). It's mainly used to embed (js, css, image..) files into assemblies and use them like physical files at runtime.

## Installation

> Most of the times you don't need to manually install this package since it comes pre-installed with the [application startup template](Startup-Templates/Application.md).

[Volo.Abp.VirtualFileSystem](https://www.nuget.org/packages/Volo.Abp.VirtualFileSystem) is the main package of the Virtual File System.

Use the ABP CLI to add this package to your project:

\* Install the [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), if you haven't installed it.

\* Open a command line (terminal) in the directory of the `.csproj` file you want to add the `Volo.Abp.VirtualFileSystem` package.

\* Run `abp add-package Volo.Abp.VirtualFileSystem` command.

If you want to do it manually, install the [Volo.Abp.VirtualFileSystem](https://www.nuget.org/packages/Volo.Abp.VirtualFileSystem) NuGet package to your project and add `[DependsOn(typeof(AbpVirtualFileSystemModule))]` to the [ABP module](Module-Development-Basics.md) class inside your project.

## Working with the Embedded Files

### Embedding the Files

A file should be first marked as **\*\*embedded resource\*\*** to embed the file into the assembly. The easiest way to do it is to select the file from the **\*\*Solution Explorer\*\*** and set **\*\*Build Action\*\*** to **\*\*Embedded Resource\*\*** from the **\*\*Properties\*\*** window. Example:

![build-action-embedded-resource-sample](images/build-action-embedded-resource-sample.png)

If you want to add multiple files, this can be tedious. Alternatively, you can directly edit your `.csproj` file:

````C#

<ItemGroup>

<EmbeddedResource Include="MyResources\\*\*\\*.\*" />

<Content Remove="MyResources\\*\*\\*.\*" />

</ItemGroup>

````

This configuration recursively adds all files under the **\*\*MyResources\*\*** folder of the project (including the files you will add in the future).

Embedding a file in the project/assembly may cause problems if a file name contains some special chars. To overcome this limitation;

1. Add [Microsoft.Extensions.FileProviders.Embedded](https://www.nuget.org/packages/Microsoft.Extensions.FileProviders.Embedded) NuGet package to the project that contains the embedded resource(s).

2. Add `<GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>` into the `<PropertyGroup>...</PropertyGroup>` section of your `.csproj` file.

> While these two steps are optional and ABP can work without these configuration, it is strongly suggested to make it.

### Configure the AbpVirtualFileSystemOptions

Use `AbpVirtualFileSystemOptions` [options class](Options.md) to register the embedded files to the virtual file system in the `ConfigureServices` method of your [module](Module-Development-Basics.md).

**\*\*Example: Add embedded files to the virtual file system\*\***

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<MyModule>();

});

````

The `AddEmbedded` extension method takes a class, finds all embedded files from the **\*\*assembly of the given class\*\*** and registers them to the virtual file system.

`AddEmbedded` can get two optional parameters;

\* `baseNamespace`: This may only needed if you didn't configure the `GenerateEmbeddedFilesManifest` step explained above and your root namespace is not empty. In this case, set your root namespace here.

\* `baseFolder`: If you don't want to expose all embedded files in the project, but only want to expose a specific folder (and sub folders/files), then you can set the base folder relative to your project root folder.

**\*\*Example: Add files under the** `MyResources` **folder in the project\*\***

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<MyModule>(

baseNamespace: "Acme.BookStore",

baseFolder: "/MyResources"

);

});

````

This example assumes;

\* Your project root (default) namespace is `Acme.BookStore`.

\* Your project has a folder, named `MyResources`

\* You only want to add `MyResources` folder to the virtual file system.

### IVirtualFileProvider

After embedding a file into an assembly and registering it to the virtual file system, the `IVirtualFileProvider` interface can be used to get the file or directory contents:

````C#

public class MyService : ITransientDependency

{

private readonly IVirtualFileProvider \_virtualFileProvider;

public MyService(IVirtualFileProvider virtualFileProvider)

{

\_virtualFileProvider = virtualFileProvider;

}

public void Test()

{

//Getting a single file

var file = \_virtualFileProvider

.GetFileInfo("/MyResources/js/test.js");

var fileContent = file.ReadAsString();

//Getting all files/directories under a directory

var directoryContents = \_virtualFileProvider

.GetDirectoryContents("/MyResources/js");

}

}

````

## ASP.NET Core Integration

The Virtual File System is well integrated to ASP.NET Core:

\* Virtual files can be used just like physical (static) files in a web application.

\* Js, css, image files and all other web content types can be embedded into assemblies and used just like the physical files.

\* An application (or another module) can **\*\*override a virtual file\*\*** of a module just like placing a file with the same name and extension into the same folder of the virtual file.

### Static Virtual File Folders

By default, ASP.NET Core only allows the `wwwroot` folder to contain the static files consumed by the clients. When you use the virtual File System, the following folders also can contain static files:

\* Pages

\* Views

\* Components

\* Themes

This allows to add `.js`, `.css`... files near to your `.cshtml` file that is easier to develop and maintain your project.

## Dealing With Embedded Files During Development

Embedding a file into an assembly and being able to use it from another project just by referencing the assembly (or adding a NuGet package) is invaluable for creating a re-usable module. However, it makes it a little bit harder to develop the module itself.

Let's assume that you're developing a module that contains an embedded JavaScript file. Whenever you change this file you must re-compile the project, re-start the application and refresh the browser page to take the change. Obviously, this is very time consuming and tedious.

What is needed is the ability for the application to directly use the physical file at development time and a browser refresh reflects any change made in the JavaScript file. The `ReplaceEmbeddedByPhysical` method makes all this possible.

The example below shows an application that depends on a module (`MyModule`) that contains embedded files. The application can access to the source code of the module at development time.

````C#

[DependsOn(typeof(MyModule))]

public class MyWebAppModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

var hostingEnvironment = context.Services.GetHostingEnvironment();

if (hostingEnvironment.IsDevelopment()) //only for development time

{

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.ReplaceEmbeddedByPhysical<MyModule>(

Path.Combine(

hostingEnvironment.ContentRootPath,

string.Format(

"..{0}MyModuleProject",

Path.DirectorySeparatorChar

)

)

);

});

}

}

}

````

The code above assumes that `MyWebAppModule` and `MyModule` are two different projects in a Visual Studio solution and `MyWebAppModule` depends on the `MyModule`.

> The [application startup template](Startup-Templates/Application.md) already uses this technique for the localization files. So, when you change a localization file it automatically detects the change.

## Replacing/Overriding Virtual Files

Virtual File System creates a unified file system on runtime, where the actual files are distributed into different modules in the development time.

If two modules adds a file to the same virtual path (like `my-path/my-file.css`), the one added later overrides/replaces the previous one ([module dependency](Module-Development-Basics.md) order determines the order of the files being added).

This feature allows your application to override/replace any virtual file defined a module that is used by your application. This is one of the fundamental extensibility features of the ABP Framework.

So, if you need to replace a file of a module, just create the file in the exactly same path in your module/application

### Physical Files

Physical files always override the virtual files. That means if you put a file under the `/wwwroot/my-folder/my-file.css`, it will override the file in the same location of the virtual file system. So, you need to know the file paths defined in the modules to override them.

# Architecture

## Modularity

### Basics

# Modularity

## Introduction

ABP Framework was designed to support to build fully modular applications and systems where every module may have entities, services, database integration, APIs, UI components and so on;

\* This document introduces the basics of the module system.

\* [Module development best practice guide](Best-Practices/Index.md) explains some **\*\*best practices\*\*** to develop **\*\*re-usable application modules\*\*** based on **\*\*DDD\*\*** principles and layers. A module designed based on this guide will be **\*\*database independent\*\*** and can be deployed as a **\*\*microservice\*\*** if needed.

\* [Pre-built application modules](Modules/Index.md) are **\*\*ready to use\*\*** in any kind of application.

\* [Module startup template](Startup-Templates/Module.md) is a jump start way to **\*\*create a new module\*\***.

\* [ABP CLI](CLI.md) has commands to support modular development.

\* All other framework features are compatible to the modularity system.

## Module Class

Every module should define a module class. The simplest way of defining a module class is to create a class derived from ``AbpModule`` as shown below:

````C#

public class BlogModule : AbpModule

{

}

````

### Configuring Dependency Injection & Other Modules

#### ConfigureServices Method

``ConfigureServices`` is the main method to add your services to the dependency injection system and configure other modules. Example:

> These methods have Async versions too, and if you want to make asynchronous calls inside these methods, override the asynchronous versions instead of the synchronous ones.

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//...

}

}

````

You can register dependencies one by one as stated in Microsoft's [documentation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection). But ABP has a **\*\*conventional dependency registration system\*\*** which automatically register all services in your assembly. See the [dependency Injection](Dependency-Injection.md) documentation for more about the dependency injection system.

You can also configure other services and modules in this way. Example:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Configure default connection string for the application

Configure<AbpDbConnectionOptions>(options =>

{

options.ConnectionStrings.Default = "......";

});

}

}

````

> `ConfigureServices` method has an asynchronous version too: `ConfigureServicesAsync`. If you want to make asynchronous calls (use the `await` keyword) inside this method, override the asynchronous version instead of the synchronous one. If you override both asynchronous and synchronous versions, only the asynchronous version will be executed.

See the [Configuration](Configuration.md) document for more about the configuration system.

#### Pre & Post Configure Services

``AbpModule`` class also defines ``PreConfigureServices`` and ``PostConfigureServices`` methods to override and write your code just before and just after ``ConfigureServices``. Notice that the code you have written into these methods will be executed before/after the ``ConfigureServices`` methods of all other modules.

> These methods have asynchronous versions too. If you want to make asynchronous calls inside these methods, override the asynchronous versions instead of the synchronous ones.

### Application Initialization

Once all the services of all modules are configured, the application starts by initializing all modules. In this phase, you can resolve services from ``IServiceProvider`` since it's ready and available.

#### OnApplicationInitialization Method

You can override ``OnApplicationInitialization`` method to execute code while application is being started.

**\*\*Example:\*\***

````C#

public class BlogModule : AbpModule

{

public override void OnApplicationInitialization(

ApplicationInitializationContext context)

{

var myService = context.ServiceProvider.GetService<MyService>();

myService.DoSomething();

}

}

````

`OnApplicationInitialization` method has an asynchronous version too. If you want to make asynchronous calls (use the `await` keyword) inside this method, override the asynchronous version instead of the synchronous one.

**\*\*Example:\*\***

````csharp

public class BlogModule : AbpModule

{

public override Task OnApplicationInitializationAsync(

ApplicationInitializationContext context)

{

var myService = context.ServiceProvider.GetService<MyService>();

await myService.DoSomethingAsync();

}

}

````

> If you override both asynchronous and synchronous versions, only the asynchronous version will be executed.

``OnApplicationInitialization`` is generally used by the startup module to construct the middleware pipeline for ASP.NET Core applications.

**\*\*Example:\*\***

````C#

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

public class AppModule : AbpModule

{

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

var env = context.GetEnvironment();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseMvcWithDefaultRoute();

}

}

````

You can also perform startup logic if your module requires it

#### Pre & Post Application Initialization

``AbpModule`` class also defines ``OnPreApplicationInitialization`` and ``OnPostApplicationInitialization`` methods to override and write your code just before and just after ``OnApplicationInitialization``. Notice that the code you have written into these methods will be executed before/after the ``OnApplicationInitialization`` methods of all other modules.

> These methods have asynchronous versions too, and if you want to make asynchronous calls inside these methods, override the asynchronous versions instead of the synchronous ones.

### Application Shutdown

Lastly, you can override ``OnApplicationShutdown`` method if you want to execute some code while application is being shutdown.

> This methods has asynchronous version too. If you want to make asynchronous calls inside this method, override the asynchronous version instead of the synchronous one.

## Module Dependencies

In a modular application, it's not unusual for one module to depend upon another module(s). An ABP module must declare a ``[DependsOn]`` attribute if it does have a dependency upon another module, as shown below:

````C#

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

[DependsOn(typeof(AbpAutofacModule))]

public class BlogModule

{

//...

}

````

You can use multiple ``DependsOn`` attribute or pass multiple module types to a single ``DependsOn`` attribute depending on your preference.

A depended module may depend on another module, but you only need to define your direct dependencies. ABP investigates the dependency graph for the application at startup and initializes/shutdowns modules in the correct order.

## Additional Module Assemblies

ABP automatically registers all the services of your module to the [dependency injection](Dependency-Injection.md) system. It finds the service types by scanning types in the assembly that defines your module class. That assembly is considered as the main assembly of your module.

Typically, every assembly contains a separate module class definition. Then modules depend on each other using the `DependsOn` attribute as explained in the previous section. However, in some rare cases, your module may consist of multiple assemblies, and only one of them defines a module class, and you want to make the other assemblies parts of your module. In that case, you can use the `AdditionalAssembly` attribute as shown below:

````csharp

[DependsOn(...)] // Your module dependencies as you normally do

[AdditionalAssembly(typeof(BlogService))] // A type in the target assembly

public class BlogModule

{

//...

}

````

In this example, we assume that the `BlogService` class is inside one assembly (`csproj`) and the `BlogModule` class is inside another assembly (`csproj`). With the `AdditionalAssembly` definition, ABP will load the assembly containing the `BlogService` class as a part of the blog module.

Notice that `BlogService` is only an arbitrary selected type in the target assembly. It is just used to indicate the related assembly. You could use any type in the assembly.

> WARNING: If you need to use the `AdditionalAssembly`, be sure that you don't design your system in a wrong way. With this example above, the `BlogService` class' assembly should normally have its own module class and the `BlogModule` should depend on it using the `DependsOn` attribute. Do not use the `AdditionalAssembly` attribute when you can already use the `DependsOn` attribute.

## Framework Modules vs Application Modules

There are **\*\*two types of modules.\*\*** They don't have any structural difference but categorized by functionality and purpose:

- **\*\*Framework modules\*\***: These are **\*\*core modules of the framework\*\*** like caching, emailing, theming, security, serialization, validation, EF Core integration, MongoDB integration... etc. They do not have application/business functionalities but makes your daily development easier by providing common infrastructure, integration and abstractions.

- **\*\*Application modules\*\***: These modules implement **\*\*specific application/business functionalities\*\*** like blogging, document management, identity management, tenant management... etc. They generally have their own entities, services, APIs and UI components. See [pre-built application modules](Modules/Index.md).

### Plug-In Modules

# Plug-In Modules

It is possible to load [modules](Module-Development-Basics.md) as plug-ins. That means you may not reference to a module's assembly in your solution, but you can load that module in the application startup just like any other module.

## Basic Usage

`IServiceCollection.AddApplication<T>()` extension method can get options to configure the plug-in sources.

**\*\*Example: Load plugins from a folder\*\***

````csharp

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.Modularity.PlugIns;

namespace MyPlugInDemo.Web

{

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddApplication<MyPlugInDemoWebModule>(options =>

{

options.PlugInSources.AddFolder(@"D:\Temp\MyPlugIns");

});

}

public void Configure(IApplicationBuilder app)

{

app.InitializeApplication();

}

}

}

````

\* This is the `Startup` class of a typical ASP.NET Core application.

\* `PlugInSources.AddFolder` gets a folder path and to load assemblies (typically `dll`s) in that folder.

That's all. ABP will discover the modules in the given folder, configure and initialize them just like regular modules.

### Plug-In Sources

`options.PlugInSources` is actually a list of `IPlugInSource` implementations and `AddFolder` is just a shortcut for the following expression:

````csharp

options.PlugInSources.Add(new FolderPlugInSource(@"D:\Temp\MyPlugIns"));

````

> `AddFolder()` only looks for the assembly file in the given folder, but not looks for the sub-folders. You can pass `SearchOption.AllDirectories` as a second parameter to explore plug-ins also from the sub-folders, recursively.

There are two more built-in Plug-In Source implementations:

\* `PlugInSources.AddFiles()` gets a list of assembly (typically `dll`) files. This is a shortcut of using `FilePlugInSource` class.

\* `PlugInSources.AddTypes()` gets a list of module class types. If you use this, you need to load the assemblies of the modules yourself, but it provides flexibility when needed. This is a shortcut of using `TypePlugInSource` class.

If you need, you can create your own `IPlugInSource` implementation and add to the `options.PlugInSources` just like the others.

## Example: Creating a Simple Plug-In

Create a simple **\*\*Class Library Project\*\*** in a solution:

![simple-plugin-library](images/simple-plugin-library.png)

You can add the ABP Framework packages that you need to use in the module. At least, you should add the `Volo.Abp.Core` package to the project, Execute the following command in the folder of the .csproj file that you want to install the package on:

````bash

abp add-package Volo.Abp.Core

````

If you haven't done it yet, you first need to install the [ABP CLI](CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Core).

Every [module](Module-Development-Basics.md) must declare a class derived from the `AbpModule`. Here, a simple module class that resolves a service and initializes it on the application startup:

````csharp

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp;

using Volo.Abp.Modularity;

namespace MyPlugIn

{

public class MyPlungInModule : AbpModule

{

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var myService = context.ServiceProvider

.GetRequiredService<MyService>();

myService.Initialize();

}

}

}

````

`MyService` can be any class registered to [Dependency Injection](Dependency-Injection.md) system, as show below:

````csharp

using Microsoft.Extensions.Logging;

using Volo.Abp.DependencyInjection;

namespace MyPlugIn

{

public class MyService : ITransientDependency

{

private readonly ILogger<MyService> \_logger;

public MyService(ILogger<MyService> logger)

{

\_logger = logger;

}

public void Initialize()

{

\_logger.LogInformation("MyService has been initialized");

}

}

}

````

Build the project, open the build folder, find the `MyPlugIn.dll`:

![simple-plug-in-dll-file](images/simple-plug-in-dll-file.png)

Copy `MyPlugIn.dll` into the plug-in folder (`D:\Temp\MyPlugIns` for this example).

> Please delete the `MyPlugIn.deps.json` file if you use `build folder` folder as `PlugInSources`.

If you have configured the main application like described above (see Basic Usage section), you should see the `MyService has been initialized` log in the application startup.

## Example: Creating a Plug-In With Razor Pages

Creating plug-ins with views inside requires a bit more attention.

> This example assumes you've [created a new web application](https://abp.io/get-started) using the application startup template and MVC / Razor Pages UI.

Create a new **\*\*Class Library\*\*** project in a solution:

![simple-razor-plugin](images/simple-razor-plugin.png)

Edit the `.csproj` file content:

````xml

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>net5.0</TargetFramework>

<OutputType>Library</OutputType>

<IsPackable>true</IsPackable>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared" Version="4.0.1" />

</ItemGroup>

</Project>

````

\* Changed `Sdk` to `Microsoft.NET.Sdk.Web`.

\* Added `OutputType` and `IsPackable` properties.

\* Added `Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared` NuGet package.

> Depending on [Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared) package is not required. You can reference to a more base package like [Volo.Abp.AspNetCore.Mvc](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc/). However, if you will build a UI page/component, it is suggested to reference to the [Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared) package since it is the most high-level package without depending on a particular [theme](UI/AspNetCore/Theming.md). If there is no problem to depend on a particular theme, you can directly reference to the theme's package to be able to use the theme-specific features in your plug-in.

Then create your module class in the plug-in:

````csharp

using System.IO;

using System.Reflection;

using Microsoft.AspNetCore.Mvc.ApplicationParts;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared;

using Volo.Abp.Modularity;

namespace MyMvcUIPlugIn

{

[DependsOn(typeof(AbpAspNetCoreMvcUiThemeSharedModule))]

public class MyMvcUIPlugInModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<IMvcBuilder>(mvcBuilder =>

{

//Add plugin assembly

mvcBuilder.PartManager.ApplicationParts.Add(new AssemblyPart(typeof(MyMvcUIPlugInModule).Assembly));

//Add CompiledRazorAssemblyPart if the PlugIn module contains razor views.

mvcBuilder.PartManager.ApplicationParts.Add(new CompiledRazorAssemblyPart(typeof(MyMvcUIPlugInModule).Assembly));

});

}

}

}

````

\* Depending on the `AbpAspNetCoreMvcUiThemeSharedModule` since we added the related NuGet package.

\* Adding the plug-in's assembly as `AssemblyPart` and `CompiledRazorAssemblyPart` to the `PartManager` of ASP.NET Core MVC. This is required by ASP.NET Core. Otherwise, your controllers or views inside the plug-in doesn't work.

You can now add a razor page, like `MyPlugInPage.cshtml` inside the `Pages` folder:

````html

@page

@model MyMvcUIPlugIn.Pages.MyPlugInPage

<h1>Welcome to my plug-in page</h1>

<p>This page is located inside a plug-in module! :)</p>

````

Now, you can build the plug-in project. It will produce the following output:

![simple-razor-plug-in-dll-file](images/simple-razor-plug-in-dll-file.png)

Copy the `MyMvcUIPlugIn.dll` into the plug-in folder (`D:\Temp\MyPlugIns` for this example).

If you have configured the main application like described above (see Basic Usage section), you should be able to visit the `/MyPlugInPage` URL when your application:

![simple-plugin-output](images/simple-plugin-output.png)

## Discussions

In real world, your plug-in may have some external dependencies. Also, your application might be designed to support plug-ins. All these are your own system requirements. What ABP does is just loading modules on the application startup. What you do inside that modules is up to you.

However, we can provide a few suggestions for some common cases.

### Library Dependencies

For package/dll dependencies, you can copy the related dlls to the plug-in folder. ABP automatically loads all assemblies in the folder and your plug-in will work as expected.

> See [Microsoft's documentation](https://docs.microsoft.com/en-us/dotnet/core/tutorials/creating-app-with-plugin-support#plugin-with-library-dependencies) for some additional explanations for that case.

### Database Schema

If your module uses a relational database and [Entity Framework Core](Entity-Framework-Core.md), it will need to have its tables available in the database. There are different ways to ensure the tables have been created when an application uses the plug-in. Some examples;

1. The Plugin may check if the database tables does exists and create the tables on the application startup or migrate them if the plug-in has been updated and requires some schema changes. You can use EF Core's migration API to do that.

2. You can improve the `DbMigrator` application to find migrations of the plug-ins and execute them.

There may be other solutions. For example, if your DB admin doesn't allow you to change the database schema in the application code, you may need to manually send a SQL file to the database admin to apply it to the database.

### Best Practices

#### Overall

## Module Development Best Practices & Conventions

### Introduction

This document describes the **\*\*best practices\*\*** and **\*\*conventions\*\*** for those who want to develop **\*\*modules\*\*** that satisfy the following specifications:

\* Develop modules that conform to the **\*\*Domain Driven Design\*\*** patterns & best practices.

\* Develop modules with **\*\*DBMS and ORM independence\*\***.

\* Develop modules that can be used as a **\*\*remote service / microservice\*\*** as well as being compatible with a **\*\*monolithic\*\*** application.

Also, this guide is mostly usable for general **\*\*application development\*\***.

### Guides

\* Overall

\* [Module Architecture](Module-Architecture.md)

\* Domain Layer

\* [Entities](Entities.md)

\* [Repositories](Repositories.md)

\* [Domain Services](Domain-Services.md)

\* Application Layer

\* [Application Services](Application-Services.md)

\* [Data Transfer Objects](Data-Transfer-Objects.md)

\* Data Access

\* [Entity Framework Core Integration](Entity-Framework-Core-Integration.md)

\* [MongoDB Integration](MongoDB-Integration.md)

## See Also

\* [E-Book: Implementing Domain Driven Design](https://abp.io/books/implementing-domain-driven-design)

#### Module Architecture

## Module Architecture Best Practices & Conventions

### Solution Structure

\* **\*\*Do\*\*** create a separated Visual Studio solution for every module.

\* **\*\*Do\*\*** name the solution as *\*CompanyName.ModuleName\** (for core ABP modules, it's *\*Volo.Abp.ModuleName\**).

\* **\*\*Do\*\*** develop the module as layered, so it has several packages (projects) those are related to each other.

\* Every package has its own module definition file and explicitly declares the dependencies for the depended packages/modules.

### Layers & Packages

The following diagram shows the packages of a well-layered module and dependencies of those packages between them:

![module-layers-and-packages](../images/module-layers-and-packages.jpg)

The ultimate goal is to allow an application to use the module in a flexible manner. Example applications:

\* **\*\*A)\*\*** A **\*\*monolithic\*\*** application;

\* Adds references to the **\*\*Web\*\*** and the **\*\*Application\*\*** packages.

\* Adds a reference to one of the **\*\*EF Core\*\*** or the **\*\*MongoDB\*\*** packages based on the preference.

\* The result;

\* The application **\*\*can show UI\*\*** of the module.

\* It hosts the **\*\*application\*\*** and **\*\*domain\*\*** layers in the **\*\*same process\*\*** (that's why it needs to have a reference to a database integration package).

\* This application also **\*\*serves\*\*** the module's **\*\*HTTP API\*\*** (since it includes the HttpApi package through the Web package).

\* **\*\*B)\*\*** An application that just serves the module as a **\*\*microservice\*\***;

\* Adds a reference to **\*\*HttpApi\*\*** and **\*\*Application\*\*** packages.

\* Adds a reference to one of the **\*\*EF Core\*\*** or the **\*\*MongoDB\*\*** packages based on the preference.

\* The result;

\* The application **\*\*can not show UI\*\*** of the module since it does not have a reference to the Web package.

\* It hosts the **\*\*application\*\*** and **\*\*domain\*\*** layers in the **\*\*same process\*\*** (that's why it needs to have a reference to a database integration package).

\* This application **\*\*serves\*\*** the module's **\*\*HTTP API\*\*** (as the main goal of the application).

\* **\*\*C)\*\*** An application that shows the module **\*\*UI\*\*** but does not host the application (just uses it as a remote service that is hosted by the application A or B);

\* Adds a reference to the **\*\*Web\*\*** and the **\*\*HttpApi.Client\*\*** packages.

\* Configures the remote endpoint for the HttpApi.Client package.

\* The result;

\* The application **\*\*can show UI\*\*** of the module.

\* It does not host the application and domain layers of the module in the same process. Instead, uses it as a **\*\*remote service\*\***.

\* This application also **\*\*serves\*\*** the module's **\*\*HTTP API\*\*** (since it includes the HttpApi package through the Web package).

\* **\*\*D)\*\*** A **\*\*client\*\*** application (or microservice) that just uses the module as a remote service (that is hosted by the application A, B or C);

\* Adds a reference to the **\*\*HttpApi.Client\*\*** package.

\* Configures the remote endpoint for the HttpApi.Client package.

\* The result;

\* The application can use all the functionality of the module as a **\*\*remote client\*\***.

\* The application is just a client and **\*\*can not serve\*\*** the **\*\*HTTP API\*\*** of the module.

\* The application is just a client and **\*\*can not show\*\*** the **\*\*UI\*\*** of the module.

\* **\*\*E\*\***) A proxy application that hosts the HTTP API of the module but just forwards all requests to another application (that is hosted by the application A, B or C);

\* Adds a reference to the **\*\*HttpApi\*\*** and **\*\*HttpApi.Client\*\*** packages.

\* Configures the remote endpoint for the HttpApi.Client package.

\* The result;

\* The application can use all the functionality of the module as a **\*\*remote client\*\***.

\* This application also **\*\*serves\*\*** the module's **\*\*HTTP API\*\***, but actually works just like a proxy by redirecting all requests (for the module) to another remote server.

Next section describes the packages in more details.

#### Domain Layer

\* **\*\*Do\*\*** divide the domain layer into two projects:

\* **\*\*Domain.Shared\*\*** package, named as *\*CompanyName.ModuleName.Domain.Shared\**, that contains constants, enums and other types those can be safely shared with the all layers of the module. This package can also be shared to 3rd-party clients. It can not contain entities, repositories, domain services or any other business objects.

\* **\*\*Domain\*\*** package, named as *\*CompanyName.ModuleName.Domain\**, that contains entities, repository interfaces, domain service interfaces and their implementations and other domain objects.

\* Domain package depends on the **\*\*Domain.Shared\*\*** package.

#### Application Layer

\* **\*\*Do\*\*** divide the application layer into two projects:

\* **\*\*Application.Contracts\*\*** package, named as *\*CompanyName.ModuleName.Application.Contracts\**, that contains application service interfaces and related data transfer objects.

\* Application contract package depends on the **\*\*Domain.Shared\*\*** package.

\* **\*\*Application\*\*** package, named as *\*CompanyName.ModuleName.Application\**, that contains application service implementations.

\* Application package depends on the **\*\*Domain\*\*** and the **\*\*Application.Contracts\*\*** packages.

#### Infrastructure Layer

\* **\*\*Do\*\*** create a separated integration package for each ORM/database integration like Entity Framework Core and MongoDB.

\* **\*\*Do\*\***, for instance, create a *\*CompanyName.ModuleName.EntityFrameworkCore\** package that abstracts the Entity Framework Core integration. ORM integration packages depend on the **\*\*Domain\*\*** package.

\* **\*\*Do not\*\*** depend on other layers from the ORM/database integration package.

\* **\*\*Do\*\*** create a separated integration package for each major library that is planned to be replaceable by another library without effecting the other packages.

#### HTTP Layer

\* **\*\*Do\*\*** create an **\*\*HTTP API\*\*** package, named as *\*CompanyName.ModuleName.HttpApi\**, to develop a REST style HTTP API for the module.

\* HTTP API package only depends on the **\*\*Application.Contracts\*\*** package. It does not depend on the Application package.

\* **\*\*Do\*\*** create a Controller for each application service (generally by implementing their interfaces). These controllers uses the application service interfaces to delegate the actions. It just configures routes, HTTP methods and other web related stuffs if needed.

\* **\*\*Do\*\*** create an **\*\*HTTP API Client\*\*** package, named as *\*CompanyName.ModuleName.HttpApi.Client\**, to provide client services for the HTTP API package. Those client services implement application interfaces as clients to a remote endpoint.

\* HTTP API Client package only depends on the **\*\*Application.Contracts\*\*** package.

\* **\*\*Do\*\*** use dynamic HTTP C# client proxy feature of the ABP framework.

#### Web Layer

\* **\*\*Do\*\*** create a **\*\*Web\*\*** package, named as *\*CompanyName.ModuleName.Web\**, that contains pages, views, scripts, styles, images and other UI components.

\* Web package only depends on the **\*\*HttpApi\*\*** package.

#### Domain Layer

##### Entities

## Entity Best Practices & Conventions

### Entities

Every aggregate root is also an entity. So, these rules are valid for aggregate roots too unless aggregate root rules override them.

- **\*\*Do\*\*** define entities in the **\*\*domain layer\*\***.

#### Primary Constructor

\* **\*\*Do\*\*** define a **\*\*primary constructor\*\*** that ensures the validity of the entity on creation. Primary constructors are used to create a new instance of the entity by the application code.

- **\*\*Do\*\*** define primary constructor as `public`, `internal` or `protected internal` based on the requirements. If it's not public, the entity is expected to be created by a domain service.

- **\*\*Do\*\*** always initialize sub collections in the primary constructor.

- **\*\*Do not\*\*** generate `Guid` keys inside the constructor. Get it as a parameter, so the calling code will use `IGuidGenerator` to generate a new `Guid` value.

#### Parameterless Constructor

- **\*\*Do\*\*** always define a `protected` parameterless constructor to be compatible with ORMs.

#### References

- **\*\*Do\*\*** always **\*\*reference\*\*** to other aggregate roots **\*\*by Id\*\***. Never add navigation properties to other aggregate roots.

#### Other Class Members

- **\*\*Do\*\*** always define properties and methods as `virtual` (except `private` methods, obviously). Because some ORMs and dynamic proxy tools require it.

- **\*\*Do\*\*** keep the entity as always **\*\*valid\*\*** and **\*\*consistent\*\*** within its own boundary.

- **\*\*Do\*\*** define properties with `private`, `protected`, `internal ` or `protected internal` setter where it is needed to protect the entity consistency and validity.

- **\*\*Do\*\*** define `public `, `internal` or `protected internal` (virtual) **\*\*methods\*\*** to change the properties (with non-public setters) if necessary.

- **\*\*Do\*\*** return the entity object (`this`) from the setter methods.

### Aggregate Roots

#### Primary Keys

\* **\*\*Do\*\*** always use a **\*\*Id\*\*** property for the aggregate root key.

\* **\*\*Do not\*\*** use **\*\*composite keys\*\*** for aggregate roots.

\* **\*\*Do\*\*** use **\*\*Guid\*\*** as the **\*\*primary key\*\*** of all aggregate roots.

#### Base Class

\* **\*\*Do\*\*** inherit from the `AggregateRoot<TKey>` or one of the audited classes (`CreationAuditedAggregateRoot<TKey>`, `AuditedAggregateRoot<TKey>` or `FullAuditedAggregateRoot<TKey>`) based on requirements.

#### Aggregate Boundary

\* **\*\*Do\*\*** keep aggregates **\*\*as small as possible\*\***. Most of the aggregates will only have primitive properties and will not have sub collections. Consider these as design decisions:

\* **\*\*Performance\*\*** & **\*\*memory\*\*** cost of loading & saving aggregates (keep in mind that an aggregate is normally loaded & saved as a single unit). Larger aggregates will consume more CPU & memory.

\* **\*\*Consistency\*\*** & **\*\*validity\*\*** boundary.

### Example

#### Aggregate Root

````C#

public class Issue : FullAuditedAggregateRoot<Guid> //Using Guid as the key/identifier

{

public virtual string Title { get; private set; } //Changed using the SetTitle() method

public virtual string Text { get; set; } //Can be directly changed. null values are allowed

public virtual Guid? MilestoneId { get; set; } //Reference to another aggregate root

public virtual bool IsClosed { get; private set; }

public virtual IssueCloseReason? CloseReason { get; private set; } //Just an enum type

public virtual Collection<IssueLabel> Labels { get; protected set; } //Sub collection

protected Issue()

{

/\* This constructor is for ORMs to be used while getting the entity from database.

\* - No need to initialize the Labels collection

since it will be overrided from the database.

- It's protected since proxying and deserialization tools

may not work with private constructors.

\*/

}

//Primary constructor

public Issue(

Guid id, //Get Guid value from the calling code

[NotNull] string title, //Indicate that the title can not be null.

string text = null,

Guid? milestoneId = null) //Optional argument

{

Id = id;

Title = Check.NotNullOrWhiteSpace(title, nameof(title)); //Validate

Text = text;

MilestoneId = milestoneId;

Labels = new Collection<IssueLabel>(); //Always initialize the collection

}

public virtual Issue SetTitle([NotNull] string title)

{

Title = Check.NotNullOrWhiteSpace(title, nameof(title)); //Validate

return this;

}

/\* AddLabel & RemoveLabel methods manages the Labels collection

\* in a safe way (prevents adding the same label twice) \*/

public virtual Issue AddLabel(Guid labelId)

{

if (Labels.Any(l => l.LabelId == labelId))

{

return;

}

Labels.Add(new IssueLabel(Id, labelId));

return this;

}

public virtual Issue RemoveLabel(Guid labelId)

{

Labels.RemoveAll(l => l.LabelId == labelId);

return this;

}

/\* Close & ReOpen methods protect the consistency

\* of the IsClosed and the CloseReason properties. \*/

public virtual void Close(IssueCloseReason reason)

{

IsClosed = true;

CloseReason = reason;

}

public virtual void ReOpen()

{

IsClosed = false;

CloseReason = null;

}

}

````

#### The Entity

````C#

public class IssueLabel : Entity

{

public virtual Guid IssueId { get; private set; }

public virtual Guid LabelId { get; private set; }

protected IssueLabel()

{

}

public IssueLabel(Guid issueId, Guid labelId)

{

IssueId = issueId;

LabelId = labelId;

}

}

````

### References

\* Effective Aggregate Design by Vaughn Vernon

http://dddcommunity.org/library/vernon\_2011

##### Repositories

## Repository Best Practices & Conventions

### Repository Interfaces

\* **\*\*Do\*\*** define repository interfaces in the **\*\*domain layer\*\***.

\* **\*\*Do\*\*** define a repository interface (like `IIdentityUserRepository`) and create its corresponding implementations for **\*\*each aggregate root\*\***.

\* **\*\*Do\*\*** always use the created repository interface from the application code.

\* **\*\*Do not\*\*** use generic repository interfaces (like `IRepository<IdentityUser, Guid>`) from the application code.

\* **\*\*Do not\*\*** use `IQueryable<TEntity>` features in the application code (domain, application... layers).

For the example aggregate root:

````C#

public class IdentityUser : AggregateRoot<Guid>

{

//...

}

````

Define the repository interface as below:

````C#

public interface IIdentityUserRepository : IBasicRepository<IdentityUser, Guid>

{

//...

}

````

\* **\*\*Do not\*\*** inherit the repository interface from the `IRepository<TEntity, TKey>` interface. Because it inherits the `IQueryable` and the repository should not expose `IQueryable` to the application.

\* **\*\*Do\*\*** inherit the repository interface from `IBasicRepository<TEntity, TKey>` (as normally) or a lower-featured interface, like `IReadOnlyRepository<TEntity, TKey>` (if it's needed).

\* **\*\*Do not\*\*** define repositories for entities those are **\*\*not aggregate roots\*\***.

### Repository Methods

\* **\*\*Do\*\*** define all repository methods as **\*\*asynchronous\*\***.

\* **\*\*Do\*\*** add an **\*\*optional\*\*** `cancellationToken` parameter to every method of the repository. Example:

````C#

Task<IdentityUser> FindByNormalizedUserNameAsync(

[NotNull] string normalizedUserName,

CancellationToken cancellationToken = default

);

````

\* **\*\*Do\*\*** add an optional `bool includeDetails = true` parameter (default value is `true`) for every repository method which returns a **\*\*single entity\*\***. Example:

````C#

Task<IdentityUser> FindByNormalizedUserNameAsync(

[NotNull] string normalizedUserName,

bool includeDetails = true,

CancellationToken cancellationToken = default

);

````

This parameter will be implemented for ORMs to eager load sub collections of the entity.

\* **\*\*Do\*\*** add an optional `bool includeDetails = false` parameter (default value is `false`) for every repository method which returns a **\*\*list of entities\*\***. Example:

````C#

Task<List<IdentityUser>> GetListByNormalizedRoleNameAsync(

string normalizedRoleName,

bool includeDetails = false,

CancellationToken cancellationToken = default

);

````

\* **\*\*Do not\*\*** create composite classes to combine entities to get from repository with a single method call. Examples: *\*UserWithRoles\**, *\*UserWithTokens\**, *\*UserWithRolesAndTokens\**. Instead, properly use `includeDetails` option to add all details of the entity when needed.

\* **\*\*Avoid\*\*** to create projection classes for entities to get less property of an entity from the repository. Example: Avoid to create BasicUserView class to select a few properties needed for the use case needs. Instead, directly use the aggregate root class. However, there may be some exceptions for this rule, where:

\* Performance is so critical for the use case and getting the whole aggregate root highly impacts the performance.

### See Also

\* [Entity Framework Core Integration](Entity-Framework-Core-Integration.md)

\* [MongoDB Integration](MongoDB-Integration.md)

##### Domain Services

## Domain Services Best Practices & Conventions

> **\*\*This document is not ready yet. Please see the [**Domain Services**](../Domain-Services.md) document.\*\***

#### Application Layer

##### Application Services

## Application Services Best Practices & Conventions

\* **\*\*Do\*\*** create an application service for each **\*\*aggregate root\*\***.

### Application Service Interface

\* **\*\*Do\*\*** define an `interface` for each application service in the **\*\*application contracts\*\*** package.

\* **\*\*Do\*\*** inherit from the `IApplicationService` interface.

\* **\*\*Do\*\*** use the `AppService` postfix for the interface name (ex: `IProductAppService`).

\* **\*\*Do\*\*** create DTOs (Data Transfer Objects) for inputs and outputs of the service.

\* **\*\*Do not\*\*** get/return entities for the service methods.

\* **\*\*Do\*\*** define DTOs based on the [DTO best practices](Data-Transfer-Objects.md).

#### Outputs

\* **\*\*Avoid\*\*** to define too many output DTOs for same or related entities. Instead, define a **\*\*basic\*\*** and a **\*\*detailed\*\*** DTO for an entity.

##### Basic DTO

**\*\*Do\*\*** define a **\*\*basic\*\*** DTO for an aggregate root.

- Include all the **\*\*primitive properties\*\*** directly on the aggregate root.

- Exception: Can **\*\*exclude\*\*** properties for **\*\*security\*\*** reasons (like `User.Password`).

- Include all the **\*\*sub collections\*\*** of the entity where every item in the collection is a simple **\*\*relation DTO\*\***.

- Inherit from one of the **\*\*extensible entity DTO\*\*** classes for aggregate roots (and entities implement the `IHasExtraProperties`).

Example:

```c#

[Serializable]

public class IssueDto : ExtensibleFullAuditedEntityDto<Guid>

{

public string Title { get; set; }

public string Text { get; set; }

public Guid? MilestoneId { get; set; }

public Collection<IssueLabelDto> Labels { get; set; }

}

[Serializable]

public class IssueLabelDto

{

public Guid IssueId { get; set; }

public Guid LabelId { get; set; }

}

```

##### Detailed DTO

**\*\*Do\*\*** define a **\*\*detailed\*\*** DTO for an entity if it has reference(s) to other aggregate roots.

\* Include all the **\*\*primitive properties\*\*** directly on the entity.

- Exception-1: Can **\*\*exclude\*\*** properties for **\*\*security\*\*** reasons (like `User.Password`).

- Exception-2: **\*\*Do\*\*** exclude reference properties (like `MilestoneId` in the example above). Will already add details for the reference properties.

\* Include a **\*\*basic DTO\*\*** property for every reference property.

\* Include all the **\*\*sub collections\*\*** of the entity where every item in the collection is the **\*\*basic DTO\*\*** of the related entity.

Example:

````C#

[Serializable]

public class IssueWithDetailsDto : ExtensibleFullAuditedEntityDto<Guid>

{

public string Title { get; set; }

public string Text { get; set; }

public MilestoneDto Milestone { get; set; }

public Collection<LabelDto> Labels { get; set; }

}

[Serializable]

public class MilestoneDto : ExtensibleEntityDto<Guid>

{

public string Name { get; set; }

public bool IsClosed { get; set; }

}

[Serializable]

public class LabelDto : ExtensibleEntityDto<Guid>

{

public string Name { get; set; }

public string Color { get; set; }

}

````

#### Inputs

\* **\*\*Do not\*\*** define any property in an input DTO that is not used in the service class.

\* **\*\*Do not\*\*** share input DTOs between application service methods.

\* **\*\*Do not\*\*** inherit an input DTO class from another one.

\* **\*\*May\*\*** inherit from an abstract base DTO class and share some properties between different DTOs in that way. However, should be very careful in that case because manipulating the base DTO would effect all related DTOs and service methods. Avoid from that as a good practice.

#### Methods

\* **\*\*Do\*\*** define service methods as asynchronous with **\*\*Async\*\*** postfix.

\* **\*\*Do not\*\*** repeat the entity name in the method names.

\* Example: Define `GetAsync(...)` instead of `GetProductAsync(...)` in the `IProductAppService`.

##### Getting A Single Entity

\* **\*\*Do\*\*** use the `GetAsync` **\*\*method name\*\***.

\* **\*\*Do\*\*** get Id with a **\*\*primitive\*\*** method parameter.

\* Return the **\*\*detailed DTO\*\***. Example:

````C#

Task<QuestionWithDetailsDto> GetAsync(Guid id);

````

##### Getting A List Of Entities

\* **\*\*Do\*\*** use the `GetListAsync` **\*\*method name\*\***.

\* **\*\*Do\*\*** get a single DTO argument for **\*\*filtering\*\***, **\*\*sorting\*\*** and **\*\*paging\*\*** if necessary.

\* **\*\*Do\*\*** implement filters optional where possible.

\* **\*\*Do\*\*** implement sorting & paging properties as optional and provide default values.

\* **\*\*Do\*\*** limit maximum page size (for performance reasons).

\* **\*\*Do\*\*** return a list of **\*\*detailed DTO\*\***s. Example:

````C#

Task<List<QuestionWithDetailsDto>> GetListAsync(QuestionListQueryDto queryDto);

````

##### Creating A New Entity

\* **\*\*Do\*\*** use the `CreateAsync` **\*\*method name\*\***.

\* **\*\*Do\*\*** get a **\*\*specialized input\*\*** DTO to create the entity.

\* **\*\*Do\*\*** inherit the DTO class from the `ExtensibleObject` (or any other class implements the `IHasExtraProperties`) to allow to pass extra properties if needed.

\* **\*\*Do\*\*** use **\*\*data annotations\*\*** for input validation.

\* Share constants between domain wherever possible (via constants defined in the **\*\*domain shared\*\*** package).

\* **\*\*Do\*\*** return **\*\*the detailed\*\*** DTO for new created entity.

\* **\*\*Do\*\*** only require the **\*\*minimum\*\*** info to create the entity but provide possibility to set others as optional properties.

Example **\*\*method\*\***:

````C#

Task<QuestionWithDetailsDto> CreateAsync(CreateQuestionDto questionDto);

````

The related **\*\*DTO\*\***:

````C#

[Serializable]

public class CreateQuestionDto : ExtensibleObject

{

[Required]

[StringLength(QuestionConsts.MaxTitleLength,

MinimumLength = QuestionConsts.MinTitleLength)]

public string Title { get; set; }

[StringLength(QuestionConsts.MaxTextLength)]

public string Text { get; set; } //Optional

public Guid? CategoryId { get; set; } //Optional

}

````

##### Updating An Existing Entity

- **\*\*Do\*\*** use the `UpdateAsync` **\*\*method name\*\***.

- **\*\*Do\*\*** get a **\*\*specialized input\*\*** DTO to update the entity.

- **\*\*Do\*\*** inherit the DTO class from the `ExtensibleObject` (or any other class implements the `IHasExtraProperties`) to allow to pass extra properties if needed.

- **\*\*Do\*\*** get the Id of the entity as a separated primitive parameter. Do not include to the update DTO.

- **\*\*Do\*\*** use **\*\*data annotations\*\*** for input validation.

- Share constants between domain wherever possible (via constants defined in the **\*\*domain shared\*\*** package).

- **\*\*Do\*\*** return **\*\*the detailed\*\*** DTO for the updated entity.

Example:

````C#

Task<QuestionWithDetailsDto> UpdateAsync(Guid id, UpdateQuestionDto updateQuestionDto);

````

##### Deleting An Existing Entity

- **\*\*Do\*\*** use the `DeleteAsync` **\*\*method name\*\***.

- **\*\*Do\*\*** get Id with a **\*\*primitive\*\*** method parameter. Example:

````C#

Task DeleteAsync(Guid id);

````

##### Other Methods

\* **\*\*Can\*\*** define additional methods to perform operations on the entity. Example:

````C#

Task<int> VoteAsync(Guid id, VoteType type);

````

This method votes a question and returns the current score of the question.

### Application Service Implementation

\* **\*\*Do\*\*** develop the application layer **\*\*completely independent from the web layer\*\***.

\* **\*\*Do\*\*** implement application service interfaces in the **\*\*application layer\*\***.

\* **\*\*Do\*\*** use the naming convention. Ex: Create `ProductAppService` class for the `IProductAppService` interface.

\* **\*\*Do\*\*** inherit from the `ApplicationService` base class.

\* **\*\*Do\*\*** make all public methods **\*\*virtual\*\***, so developers may inherit and override them.

\* **\*\*Do not\*\*** make **\*\*private\*\*** methods. Instead make them **\*\*protected virtual\*\***, so developers may inherit and override them.

#### Using Repositories

\* **\*\*Do\*\*** use the specifically designed repositories (like `IProductRepository`).

\* **\*\*Do not\*\*** use generic repositories (like `IRepository<Product>`).

#### Querying Data

\* **\*\*Do not\*\*** use LINQ/SQL for querying data from database inside the application service methods. It's repository's responsibility to perform LINQ/SQL queries from the data source.

#### Extra Properties

\* **\*\*Do\*\*** use either `MapExtraPropertiesTo` extension method ([see](../Object-Extensions.md)) or configure the object mapper (`MapExtraProperties`) to allow application developers to be able to extend the objects and services.

#### Manipulating / Deleting Entities

\* **\*\*Do\*\*** always get all the related entities from repositories to perform the operations on them.

\* **\*\*Do\*\*** call repository's Update/UpdateAsync method after updating an entity. Because, not all database APIs support change tracking & auto update.

#### Handle files

\* **\*\*Do not\*\*** use any web components like `IFormFile` or `Stream` in the application services. If you want to serve a file you can use `byte[]`.

\* **\*\*Do\*\*** use a `Controller` to handle file uploading then pass the `byte[]` of the file to the application service method.

#### Using Other Application Services

\* **\*\*Do not\*\*** use other application services of the same module/application. Instead;

\* Use domain layer to perform the required task.

\* Extract a new class and share between the application services to accomplish the code reuse when necessary. But be careful to don't couple two use cases. They may seem similar at the beginning, but may evolve to different directions by time. So, use code sharing carefully.

\* **\*\*Can\*\*** use application services of others only if;

\* They are parts of another module / microservice.

\* The current module has only reference to the application contracts of the used module.

##### Data Transfer Objects

## Data Transfer Objects Best Practices & Conventions

\* **\*\*Do\*\*** define DTOs in the **\*\*application contracts\*\*** package.

\* **\*\*Do\*\*** inherit from the pre-built **\*\*base DTO classes\*\*** where possible and necessary (like `EntityDto<TKey>`, `CreationAuditedEntityDto<TKey>`, `AuditedEntityDto<TKey>`, `FullAuditedEntityDto<TKey>` and so on).

\* **\*\*Do\*\*** inherit from the **\*\*extensible DTO\*\*** classes for the **\*\*aggregate roots\*\*** (like `ExtensibleAuditedEntityDto<TKey>`), because aggregate roots are extensible objects and extra properties are mapped to DTOs in this way.

\* **\*\*Do\*\*** define DTO members with **\*\*public getter and setter\*\***.

\* **\*\*Do\*\*** use **\*\*data annotations\*\*** for **\*\*validation\*\*** on the properties of DTOs those are inputs of the service.

\* **\*\*Do\*\*** not add any **\*\*logic\*\*** into DTOs except implementing `IValidatableObject` when necessary.

\* **\*\*Do\*\*** mark all DTOs as **\*\*[Serializable]\*\*** since they are already serializable and developers may want to binary serialize them.

#### Data Access

##### Entity Framework Core Integration

## Entity Framework Core Integration Best Practices

> See [Entity Framework Core Integration document](../Entity-Framework-Core.md) for the basics of the EF Core integration.

- **\*\*Do\*\*** define a separated `DbContext` interface and class for each module.

- **\*\*Do not\*\*** rely on lazy loading on the application development.

- **\*\*Do not\*\*** enable lazy loading for the `DbContext`.

### DbContext Interface

- **\*\*Do\*\*** define an **\*\*interface\*\*** for the `DbContext` that inherits from `IEfCoreDbContext`.

- **\*\*Do\*\*** add a `ConnectionStringName` **\*\*attribute\*\*** to the `DbContext` interface.

- **\*\*Do\*\*** add `DbSet<TEntity>` **\*\*properties\*\*** to the `DbContext` interface for only aggregate roots. Example:

````C#

[ConnectionStringName("AbpIdentity")]

public interface IIdentityDbContext : IEfCoreDbContext

{

DbSet<IdentityUser> Users { get; }

DbSet<IdentityRole> Roles { get; }

}

````

\* **\*\*Do not\*\*** define `set;` for the properties in this interface.

### DbContext class

\* **\*\*Do\*\*** inherit the `DbContext` from the `AbpDbContext<TDbContext>` class.

\* **\*\*Do\*\*** add a `ConnectionStringName` attribute to the `DbContext` class.

\* **\*\*Do\*\*** implement the corresponding `interface` for the `DbContext` class. Example:

````C#

[ConnectionStringName("AbpIdentity")]

public class IdentityDbContext : AbpDbContext<IdentityDbContext>, IIdentityDbContext

{

public DbSet<IdentityUser> Users { get; set; }

public DbSet<IdentityRole> Roles { get; set; }

public IdentityDbContext(DbContextOptions<IdentityDbContext> options)

: base(options)

{

}

//code omitted for brevity

}

````

### Table Prefix and Schema

- **\*\*Do\*\*** add static `TablePrefix` and `Schema` **\*\*properties\*\*** to the `DbContext` class. Set default value from a constant. Example:

````C#

public static string TablePrefix { get; set; } = AbpIdentityConsts.DefaultDbTablePrefix;

public static string Schema { get; set; } = AbpIdentityConsts.DefaultDbSchema;

````

- **\*\*Do\*\*** always use a short `TablePrefix` value for a module to create **\*\*unique table names\*\*** in a shared database. `Abp` table prefix is reserved for ABP core modules.

- **\*\*Do\*\*** set `Schema` to `null` as default.

### Model Mapping

- **\*\*Do\*\*** explicitly **\*\*configure all entities\*\*** by overriding the `OnModelCreating` method of the `DbContext`. Example:

````C#

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

builder.ConfigureIdentity();

}

````

- **\*\*Do not\*\*** configure model directly in the `OnModelCreating` method. Instead, create an **\*\*extension method\*\*** for `ModelBuilder`. Use Configure*\*ModuleName\** as the method name. Example:

````C#

public static class IdentityDbContextModelBuilderExtensions

{

public static void ConfigureIdentity([NotNull] this ModelBuilder builder)

{

Check.NotNull(builder, nameof(builder));

builder.Entity<IdentityUser>(b =>

{

b.ToTable(AbpIdentityDbProperties.DbTablePrefix + "Users", AbpIdentityDbProperties.DbSchema);

b.ConfigureByConvention();

//code omitted for brevity

});

builder.Entity<IdentityUserClaim>(b =>

{

b.ToTable(AbpIdentityDbProperties.DbTablePrefix + "UserClaims", AbpIdentityDbProperties.DbSchema);

b.ConfigureByConvention();

//code omitted for brevity

});

//code omitted for brevity

}

}

````

\* **\*\*Do\*\*** call `b.ConfigureByConvention();` for each entity mapping (as shown above).

### Repository Implementation

- **\*\*Do\*\*** **\*\*inherit\*\*** the repository from the `EfCoreRepository<TDbContext, TEntity, TKey>` class and implement the corresponding repository interface. Example:

````C#

public class EfCoreIdentityUserRepository

: EfCoreRepository<IIdentityDbContext, IdentityUser, Guid>, IIdentityUserRepository

{

public EfCoreIdentityUserRepository(

IDbContextProvider<IIdentityDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

````

\* **\*\*Do\*\*** use the `DbContext` interface as the generic parameter, not the class.

\* **\*\*Do\*\*** pass the `cancellationToken` to EF Core using the `GetCancellationToken` helper method. Example:

````C#

public virtual async Task<IdentityUser> FindByNormalizedUserNameAsync(

string normalizedUserName,

bool includeDetails = true,

CancellationToken cancellationToken = default)

{

return await (await GetDbSetAsync())

.IncludeDetails(includeDetails)

.FirstOrDefaultAsync(

u => u.NormalizedUserName == normalizedUserName,

GetCancellationToken(cancellationToken)

);

}

````

`GetCancellationToken` fallbacks to the `ICancellationTokenProvider.Token` to obtain the cancellation token if it is not provided by the caller code.

- **\*\*Do\*\*** create a `IncludeDetails` **\*\*extension method\*\*** for the `IQueryable<TEntity>` for each aggregate root which has **\*\*sub collections\*\***. Example:

````C#

public static IQueryable<IdentityUser> IncludeDetails(

this IQueryable<IdentityUser> queryable,

bool include = true)

{

if (!include)

{

return queryable;

}

return queryable

.Include(x => x.Roles)

.Include(x => x.Logins)

.Include(x => x.Claims)

.Include(x => x.Tokens);

}

````

\* **\*\*Do\*\*** use the `IncludeDetails` extension method in the repository methods just like used in the example code above (see `FindByNormalizedUserNameAsync`).

- **\*\*Do\*\*** override `WithDetails` method of the repository for aggregates root which have **\*\*sub collections\*\***. Example:

````C#

public override async Task<IQueryable<IdentityUser>> WithDetailsAsync()

{

// Uses the extension method defined above

return (await GetQueryableAsync()).IncludeDetails();

}

````

### Module Class

- **\*\*Do\*\*** define a module class for the Entity Framework Core integration package.

- **\*\*Do\*\*** add `DbContext` to the `IServiceCollection` using the `AddAbpDbContext<TDbContext>` method.

- **\*\*Do\*\*** add implemented repositories to the options for the `AddAbpDbContext<TDbContext>` method. Example:

````C#

[DependsOn(

typeof(AbpIdentityDomainModule),

typeof(AbpEntityFrameworkCoreModule)

)]

public class AbpIdentityEntityFrameworkCoreModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAbpDbContext<IdentityDbContext>(options =>

{

options.AddRepository<IdentityUser, EfCoreIdentityUserRepository>();

options.AddRepository<IdentityRole, EfCoreIdentityRoleRepository>();

});

}

}

````

##### MongoDB Integration

## MongoDB Integration

\* Do define a separated `MongoDbContext` interface and class for each module.

### MongoDbContext Interface

- **\*\*Do\*\*** define an **\*\*interface\*\*** for the `MongoDbContext` that inherits from `IAbpMongoDbContext`.

- **\*\*Do\*\*** add a `ConnectionStringName` **\*\*attribute\*\*** to the `MongoDbContext` interface.

- **\*\*Do\*\*** add `IMongoCollection<TEntity>` **\*\*properties\*\*** to the `MongoDbContext` interface only for the aggregate roots. Example:

````C#

[ConnectionStringName("AbpIdentity")]

public interface IAbpIdentityMongoDbContext : IAbpMongoDbContext

{

IMongoCollection<IdentityUser> Users { get; }

IMongoCollection<IdentityRole> Roles { get; }

}

````

### MongoDbContext class

- **\*\*Do\*\*** inherit the `MongoDbContext` from the `AbpMongoDbContext` class.

- **\*\*Do\*\*** add a `ConnectionStringName` attribute to the `MongoDbContext` class.

- **\*\*Do\*\*** implement the corresponding `interface` for the `MongoDbContext` class. Example:

```c#

[ConnectionStringName("AbpIdentity")]

public class AbpIdentityMongoDbContext : AbpMongoDbContext, IAbpIdentityMongoDbContext

{

public IMongoCollection<IdentityUser> Users => Collection<IdentityUser>();

public IMongoCollection<IdentityRole> Roles => Collection<IdentityRole>();

//code omitted for brevity

}

```

### Collection Prefix

- **\*\*Do\*\*** add static `CollectionPrefix` **\*\*property\*\*** to the `DbContext` class. Set default value from a constant. Example:

```c#

public static string CollectionPrefix { get; set; } = AbpIdentityConsts.DefaultDbTablePrefix;

```

Used the same constant defined for the EF Core integration table prefix in this example.

- **\*\*Do\*\*** always use a short `CollectionPrefix` value for a module to create **\*\*unique collection names\*\*** in a shared database. `Abp` collection prefix is reserved for ABP core modules.

### Collection Mapping

- **\*\*Do\*\*** explicitly **\*\*configure all aggregate roots\*\*** by overriding the `CreateModel` method of the `MongoDbContext`. Example:

```c#

protected override void CreateModel(IMongoModelBuilder modelBuilder)

{

base.CreateModel(modelBuilder);

modelBuilder.ConfigureIdentity();

}

```

- **\*\*Do not\*\*** configure model directly in the `CreateModel` method. Instead, create an **\*\*extension method\*\*** for the `IMongoModelBuilder`. Use Configure*\*ModuleName\** as the method name. Example:

```c#

public static class AbpIdentityMongoDbContextExtensions

{

public static void ConfigureIdentity(

this IMongoModelBuilder builder,

Action<IdentityMongoModelBuilderConfigurationOptions> optionsAction = null)

{

Check.NotNull(builder, nameof(builder));

builder.Entity<IdentityUser>(b =>

{

b.CollectionName = AbpIdentityDbProperties.DbTablePrefix + "Users";

});

builder.Entity<IdentityRole>(b =>

{

b.CollectionName = AbpIdentityDbProperties.DbTablePrefix + "Roles";

});

}

}

```

### Repository Implementation

- **\*\*Do\*\*** **\*\*inherit\*\*** the repository from the `MongoDbRepository<TMongoDbContext, TEntity, TKey>` class and implement the corresponding repository interface. Example:

```c#

public class MongoIdentityUserRepository

: MongoDbRepository<IAbpIdentityMongoDbContext, IdentityUser, Guid>,

IIdentityUserRepository

{

public MongoIdentityUserRepository(

IMongoDbContextProvider<IAbpIdentityMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

```

- **\*\*Do\*\*** pass the `cancellationToken` to the MongoDB Driver using the `GetCancellationToken` helper method. Example:

```c#

public async Task<IdentityUser> FindByNormalizedUserNameAsync(

string normalizedUserName,

bool includeDetails = true,

CancellationToken cancellationToken = default)

{

return await (await GetMongoQueryableAsync())

.FirstOrDefaultAsync(

u => u.NormalizedUserName == normalizedUserName,

GetCancellationToken(cancellationToken)

);

}

```

`GetCancellationToken` fallbacks to the `ICancellationTokenProvider.Token` to obtain the cancellation token if it is not provided by the caller code.

\* **\*\*Do\*\*** ignore the `includeDetails` parameters for the repository implementation since MongoDB loads the aggregate root as a whole (including sub collections) by default.

\* **\*\*Do\*\*** use the `GetMongoQueryableAsync()` method to obtain an `IQueryable<TEntity>` to perform queries wherever possible. Because;

\* `GetMongoQueryableAsync()` method automatically uses the `ApplyDataFilters` method to filter the data based on the current data filters (like soft delete and multi-tenancy).

\* Using `IQueryable<TEntity>` makes the code as much as similar to the EF Core repository implementation and easy to write and read.

\* **\*\*Do\*\*** implement data filtering if it is not possible to use the `GetMongoQueryable()` method.

### Module Class

- **\*\*Do\*\*** define a module class for the MongoDB integration package.

- **\*\*Do\*\*** add `MongoDbContext` to the `IServiceCollection` using the `AddMongoDbContext<TMongoDbContext>` method.

- **\*\*Do\*\*** add implemented repositories to the options for the `AddMongoDbContext<TMongoDbContext>` method. Example:

```c#

[DependsOn(

typeof(AbpIdentityDomainModule),

typeof(AbpUsersMongoDbModule)

)]

public class AbpIdentityMongoDbModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddMongoDbContext<AbpIdentityMongoDbContext>(options =>

{

options.AddRepository<IdentityUser, MongoIdentityUserRepository>();

options.AddRepository<IdentityRole, MongoIdentityRoleRepository>();

});

}

}

```

Notice that this module class also calls the static `BsonClassMap` configuration method defined above.

### Customizing/Extending Modules

#### Overall

# Customizing the Existing Modules

ABP Framework has been designed to support to build fully [modular applications](Module-Development-Basics.md) and systems. It also provides some [pre-built application modules](Modules/Index.md) those are **\*\*ready to use\*\*** in any kind of application.

For example, you can **\*\*re-use\*\*** the [Identity Management Module](Modules/Identity.md) to add user, role and permission management to your application. The [application startup template](Startup-Templates/Application.md) already comes with Identity and some other modules **\*\*pre-installed\*\***.

## Re-Using an Application Module

You have two options to re-use an application module.

### As Package References

You can add **\*\*NuGet\*\*** & **\*\*NPM\*\*** package references of the related module to your application and configure the module (based on its documentation) to integrate to your application.

As mentioned before, the [application startup template](Startup-Templates/Application.md) already comes with some **\*\*fundamental modules pre-installed\*\***. It uses the modules as NuGet & NPM package references.

This approach has the following benefits:

\* Your solution will be **\*\*clean\*\*** and only contains your **\*\*own application code\*\***.

\* You can **\*\*easily upgrade\*\*** a module when a new version is available. `abp update` [CLI](CLI.md) command makes it even easier. In this way, you can continue to get **\*\*new features and bug fixes\*\***.

However, there is a drawback:

\* You may not able to **\*\*customize\*\*** the module because the module source is not in your solution.

This document explains **\*\*how to customize or extend\*\*** a depended module without need to change its source code. While it is limited compared to a full source code change opportunity, there are still some good ways to make some customizations.

If you don't think to make huge changes on the pre-built modules, re-using them as package reference is the recommended way.

### Including the Source Code

If you want to make **\*\*huge changes\*\*** or add **\*\*major features\*\*** on a pre-built module, but the available extension points are not enough, you can consider to directly work the source code of the depended module.

In this case, you typically **\*\*add the source code\*\*** of the module to your solution and replace **\*\*every\*\*** package reference in the solution with its corresponding local project references. **\*\*[**ABP CLI**](CLI.md)\*\***'s `add-module` command automates this process for you with the `--with-source-code` parameter. This command can also replace a module by its source code if the module already installed as NuGet packages.

#### Separating the Module Solution

You may prefer to not include the module source code **\*\*directly into your solution\*\***. Every module consists of 10+ project files and adding **\*\*multiple modules\*\*** may impact on the **\*\*size\*\*** of your solution **\*\*load & development time.\*\*** Also, you may have different development teams working on different modules, so you don't want to make the module code available to the application development team.

In any case, you can create a **\*\*separate solution\*\*** for the desired module and depend on the module as project references out of the solution. We do it like that for the [abp repository](https://github.com/abpframework/abp/).

> One problem we see is Visual Studio doesn't play nice with this kind of approach (it doesn't support well to have references to local projects out of the solution directory). If you get error while building the application (depends on an external module), run `dotnet restore` in the command line after opening the application's solution in the Visual Studio.

#### Publishing the Customized Module as Packages

One alternative scenario could be re-packaging the module source code (as NuGet/NPM packages) and using as package references. You can use a local private NuGet/NPM server for your company, for example.

## Module Customization / Extending Approaches

This section suggests some approaches if you decided to use pre-built application modules as NuGet/NPM package references. The following documents explain how to customize/extend existing modules in different ways.

### Module Entity Extension System

> Module entity extension system is the **\*\*main and high level extension system\*\*** that allows you to **\*\*define new properties\*\*** for existing entities of the depended modules. It automatically **\*\*adds properties to the entity, database, HTTP API and the user interface\*\*** in a single point.

See the [Module Entity Extensions document](Module-Entity-Extensions.md) to learn how to use it.

### Extending Entities

If you only need to get/set extra data on an existing entity, follow the [Extending Entities](Customizing-Application-Modules-Extending-Entities.md) document.

### Overriding Services/Components

In addition to the extensibility systems, you can partially or completely override any service or user interface page/component.

\* [Overriding Services](Customizing-Application-Modules-Overriding-Services.md)

\* [Overriding the User Interface](Customizing-Application-Modules-Overriding-User-Interface.md)

### Additional UI Extensibility Points

There are some low level systems that you can control entity actions, table columns and page toolbar of a page defined by a module.

#### Entity Actions

Entity action extension system allows you to add a new action to the action menu for an entity on the user interface;

\* [Entity Action Extensions for ASP.NET Core UI](UI/AspNetCore/Entity-Action-Extensions.md)

\* [Entity Action Extensions for Blazor UI](UI/Blazor/Entity-Action-Extensions.md)

\* [Entity Action Extensions for Angular](UI/Angular/Entity-Action-Extensions.md)

#### Data Table Column Extensions

Data table column extension system allows you to add a new column in the data table on the user interface;

\* [Data Table Column Extensions for ASP.NET Core UI](UI/AspNetCore/Data-Table-Column-Extensions.md)

\* [Data Table Column Extensions for Blazor UI](UI/Blazor/Data-Table-Column-Extensions.md)

\* [Data Table Column Extensions for Angular](UI/Angular/Data-Table-Column-Extensions.md)

#### Page Toolbar

Page toolbar system allows you to add components to the toolbar of a page;

\* [Page Toolbar Extensions for ASP.NET Core UI](UI/AspNetCore/Page-Toolbar-Extensions.md)

\* [Page Toolbar Extensions for Blazor UI](UI/Blazor/Page-Toolbar-Extensions.md)

\* [Page Toolbar Extensions for Angular](UI/Angular/Page-Toolbar-Extensions.md)

#### Others

\* [Dynamic Form Extensions for Angular](UI/Angular/Dynamic-Form-Extensions.md)

## See Also

Also, see the following documents:

\* See [the localization document](Localization.md) to learn how to extend existing localization resources.

\* See [the settings document](Settings.md) to learn how to change setting definitions of a depended module.

\* See [the authorization document](Authorization.md) to learn how to change permission definitions of a depended module.

#### Module Entity Extension System

# Module Entity Extensions

Module entity extension system is a **\*\*high level\*\*** extension system that allows you to **\*\*define new properties\*\*** for existing entities of the depended modules. It automatically **\*\*adds properties to the entity, database, HTTP API and the user interface\*\*** in a single point.

> The module must be developed the *\*Module Entity Extensions\** system in mind. All the **\*\*official modules\*\*** supports this system wherever possible.

## Quick Example

Open the `YourProjectNameModuleExtensionConfigurator` class inside the `Domain.Shared` project of your solution and change the `ConfigureExtraProperties`method as shown below to add a `SocialSecurityNumber` property to the `IdentityUser` entity of the [Identity Module](Modules/Identity.md).

````csharp

public static void ConfigureExtraProperties()

{

OneTimeRunner.Run(() =>

{

ObjectExtensionManager.Instance.Modules()

.ConfigureIdentity(identity =>

{

identity.ConfigureUser(user =>

{

user.AddOrUpdateProperty<string>( //property type: string

"SocialSecurityNumber", //property name

property =>

{

//validation rules

property.Attributes.Add(new RequiredAttribute());

property.Attributes.Add(

new StringLengthAttribute(64) {

MinimumLength = 4

}

);

//...other configurations for this property

}

);

});

});

});

}

````

>This method is called inside the `YourProjectNameDomainSharedModule` at the beginning of the application. `OneTimeRunner` is a utility class that guarantees to execute this code only one time per application, since multiple calls are unnecessary.

\* `ObjectExtensionManager.Instance.Modules()` is the starting point to configure a module. `ConfigureIdentity(...)` method is used to configure the entities of the Identity Module.

\* `identity.ConfigureUser(...)` is used to configure the user entity of the identity module. Not all entities are designed to be extensible (since it is not needed). Use the intellisense to discover the extensible modules and entities.

\* `user.AddOrUpdateProperty<string>(...)` is used to add a new property to the user entity with the `string` type (`AddOrUpdateProperty` method can be called multiple times for the same property of the same entity. Each call can configure the options of the same property, but only one property is added to the entity with the same property name). You can call this method with different property names to add more properties.

\* `SocialSecurityNumber` is the name of the new property.

\* `AddOrUpdateProperty` gets a second argument (the `property =>` lambda expression) to configure additional options for the new property.

\* We can add data annotation attributes like shown here, just like adding a data annotation attribute to a class property.

#### Create & Update Forms

Once you define a property, it appears in the create and update forms of the related entity:

![add-new-property-to-user-form](images/add-new-property-to-user-form.png)

`SocialSecurityNumber` field comes into the form. Next sections will explain the localization and the validation for this new property.

### Data Table

New properties also appear in the data table of the related page:

![add-new-property-to-user-form](images/add-new-property-to-user-table.png)

`SocialSecurityNumber` column comes into the table. Next sections will explain the option to hide this column from the data table.

## Property Options

There are some options that you can configure while defining a new property.

### Display Name

You probably want to set a different (human readable) display name for the property that is shown on the user interface.

#### Don't Want to Localize?

If your application is not localized, you can directly set the `DisplayName` for the property to a `FixedLocalizableString` object. Example:

````csharp

property =>

{

property.DisplayName = new FixedLocalizableString("Social security no");

}

````

#### Localizing the Display Name

If you want to localize the display name, you have two options.

##### Localize by Convention

Instead of setting the `property.DisplayName`, you can directly open your localization file (like `en.json`) and add the following entry to the `texts` section:

````json

"SocialSecurityNumber": "Social security no"

````

Define the same `SocialSecurityNumber` key (the property name you've defined before) in your localization file for each language you support. That's all!

In some cases, the localization key may conflict with other keys in your localization files. In such cases, you can use the `DisplayName:` prefix for display names in the localization file (`DisplayName:SocialSecurityNumber` as the localization key for this example). Extension system looks for prefixed version first, then fallbacks to the non prefixed name (it then fallbacks to the property name if you haven't localized it).

> This approach is recommended since it is simple and suitable for most scenarios.

##### Localize using the `DisplayName` Property

If you want to specify the localization key or the localization resource, you can still set the `DisplayName` option:

````csharp

property =>

{

property.DisplayName =

LocalizableString.Create<MyProjectNameResource>(

"UserSocialSecurityNumberDisplayName"

);

}

````

\* `MyProjectNameResource` is the localization resource and `UserSocialSecurityNumberDisplayName` is the localization key in the localization resource.

> See [the localization document](Localization.md) if you want to learn more about the localization system.

#### Default Value

A default value is automatically set for the new property, which is the natural default value for the property type, like `null` for `string`, `false` for `bool` or `0` for `int`.

There are two ways to override the default value:

##### DefaultValue Option

`DefaultValue` option can be set to any value:

````csharp

property =>

{

property.DefaultValue = 42;

}

````

##### DefaultValueFactory Options

`DefaultValueFactory` can be set to a function that returns the default value:

````csharp

property =>

{

property.DefaultValueFactory = () => DateTime.Now;

}

````

`options.DefaultValueFactory` has a higher priority than the `options.DefaultValue` .

> Tip: Use `DefaultValueFactory` option only if the default value may change over the time (like `DateTime.Now` in this example). If it is a constant value, then use the `DefaultValue` option.

### Validation

Entity extension system allows you to define validation for extension properties in a few ways.

#### Data Annotation Attributes

`Attributes` is a list of attributes associated to this property. The example code below adds two [data annotation validation attributes](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation) to the property:

````csharp

property =>

{

property.Attributes.Add(new RequiredAttribute());

property.Attributes.Add(new StringLengthAttribute(64) {MinimumLength = 4});

}

````

When you run the application, you see that the validation works out of the box:

![add-new-propert-to-user-form](images/add-new-property-to-user-form-validation-error.png)

Since we've added the `RequiredAttribute`, it doesn't allow to left it blank. The validation system works;

\* On the user interface (with automatic localization).

\* On the HTTP API. Even if you directly perform an HTTP request, you get validation errors with a proper HTTP status code.

\* On the `SetProperty(...)` method on the entity (see [the document](Entities.md) if you wonder what is the `SetProperty()` method).

So, it automatically makes a full stack validation.

> See the [ASP.NET Core MVC Validation document](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation) to learn more about the attribute based validation.

##### Default Validation Attributes

There are some attributes **\*\*automatically added\*\*** when you create certain type of properties;

\* `RequiredAttribute` is added for **\*\*non nullable\*\*** primitive property types (e.g. `int`, `bool`, `DateTime`...) and `enum` types. If you want to allow nulls, make the property nullable (e.g. `int?`).

\* `EnumDataTypeAttribute` is added for **\*\*enum types\*\***, to prevent to set invalid enum values.

Use `property.Attributes.Clear();` if you don't want these attributes.

#### Validation Actions

Validation actions allows you to execute a custom code to perform the validation. The example below checks if the `SocialSecurityNumber` starts with `B` and adds a validation error if so:

````csharp

property =>

{

property.Attributes.Add(new RequiredAttribute());

property.Attributes.Add(new StringLengthAttribute(64) {MinimumLength = 4});

property.Validators.Add(context =>

{

if (((string) context.Value).StartsWith("B"))

{

context.ValidationErrors.Add(

new ValidationResult(

"Social security number can not start with the letter 'B', sorry!",

new[] {"extraProperties.SocialSecurityNumber"}

)

);

}

});

}

````

Using a `RegularExpressionAttribute` might be better in this case, but this is just an example. Anyway, if you enter a value starts with the letter `B` you get the following error **\*\*while saving the form\*\***:

![add-new-propert-to-user-form](images/add-new-property-to-user-form-validation-error-custom.png)

##### The Context Object

The `context` object has useful properties that can be used in your custom validation action. For example, you can use the `context.ServiceProvider` to resolve services from the [dependency injection system](Dependency-Injection.md). The example below gets the localizer and adds a localized error message:

````csharp

if (((string) context.Value).StartsWith("B"))

{

var localizer = context.ServiceProvider

.GetRequiredService<IStringLocalizer<MyProjectNameResource>>();

context.ValidationErrors.Add(

new ValidationResult(

localizer["SocialSecurityNumberCanNotStartWithB"],

new[] {"extraProperties.SocialSecurityNumber"}

)

);

}

````

>`context.ServiceProvider` is nullable! It can be `null` only if you use the `SetProperty(...)` method on the object. Because DI system is not available on this time. While this is a rare case, you should perform a fallback logic when `context.ServiceProvider` is `null`. For this example, you would add a non-localized error message. This is not a problem since setting an invalid value to a property generally is a programmer mistake and you mostly don't need to localization in this case. In any way, you would not be able to use localization even in a regular property setter. But, if you are serious about localization, you can throw a business exception (see the [exception handling document](https://docs.abp.io/en/abp/latest/Exception-Handling) to learn how to localize a business exception).

### UI Visibility

When you define a property, it appears on the data table, create and edit forms on the related UI page. However, you can control each one individually. Example:

````csharp

property =>

{

property.UI.OnTable.IsVisible = false;

//...other configurations

}

````

Use `property.UI.OnCreateForm` and `property.UI.OnEditForm` to control forms too. If a property is required, but not added to the create form, you definitely get a validation exception, so use this option carefully. But a required property may not be in the edit form if that's your requirement.

### UI Order

When you define a property, it appears on the data table, create and edit forms on the related UI page. However, you can control its order. Example:

````csharp

property =>

{

property.UI.Order = 1;

//...other configurations

}

````

Use `property.UI.OnCreateForm` and `property.UI.OnEditForm` to control forms too. If a property is required, but not added to the create form, you definitely get a validation exception, so use this option carefully. But a required property may not be in the edit form if that's your requirement.

### HTTP API Availability

Even if you disable a property on UI, it can be still available through the HTTP API. By default, a property is available on all APIs.

Use the `property.Api` options to make a property unavailable in some API endpoints.

````csharp

property =>

{

property.Api.OnUpdate.IsAvailable = false;

}

````

In this example, Update HTTP API will not allow to set a new value to this property. In this case, you also want to disable this property on the edit form:

````csharp

property =>

{

property.Api.OnUpdate.IsAvailable = false;

property.UI.OnEditForm.IsVisible = false;

}

````

In addition to the `property.Api.OnUpdate`, you can set `property.Api.OnCreate` and `property.Api.OnGet` for a fine control the API endpoint.

## Special Types

### Enum

Module extension system naturally supports the `enum` types.

An example enum type:

````csharp

public enum UserType

{

Regular,

Moderator,

SuperUser

}

````

You can add enum properties just like others:

````csharp

user.AddOrUpdateProperty<UserType>("Type");

````

An enum properties is shown as combobox (select) in the create/edit forms:

![add-new-property-enum](images/add-new-property-enum.png)

#### Localization

Enum member name is shown on the table and forms by default. If you want to localize it, just create a new entry on your [localization](https://docs.abp.io/en/abp/latest/Localization) file:

````json

"Enum:UserType.0": "Super user"

````

One of the following names can be used as the localization key:

\* `Enum:UserType.0`

\* `Enum:UserType.SuperUser`

\* `UserType.0`

\* `UserType.SuperUser`

\* `SuperUser`

Localization system searches for the key with the given order. Localized text are used on the table and the create/edit forms.

### Navigation Properties / Foreign Keys

It is supported to add an extension property to an entity that is Id of another entity (foreign key).

#### Example: Associate a department to a user

````csharp

ObjectExtensionManager.Instance.Modules()

.ConfigureIdentity(identity =>

{

identity.ConfigureUser(user =>

{

user.AddOrUpdateProperty<Guid>(

"DepartmentId",

property =>

{

property.UI.Lookup.Url = "/api/departments";

property.UI.Lookup.DisplayPropertyName = "name";

}

);

});

});

````

`UI.Lookup.Url` option takes a URL to get list of departments to select on edit/create forms. This endpoint can be a typical controller, an [auto API controller](API/Auto-API-Controllers.md) or any type of endpoint that returns a proper JSON response.

An example implementation that returns a fixed list of departments (in real life, you get the list from a data source):

````csharp

[Route("api/departments")]

public class DepartmentController : AbpController

{

[HttpGet]

public async Task<ListResultDto<DepartmentDto>> GetAsync()

{

return new ListResultDto<DepartmentDto>(

new[]

{

new DepartmentDto

{

Id = Guid.Parse("6267f0df-870f-4173-be44-d74b4b56d2bd"),

Name = "Human Resources"

},

new DepartmentDto

{

Id = Guid.Parse("21c7b61f-330c-489e-8b8c-80e0a78a5cc5"),

Name = "Production"

}

}

);

}

}

````

This API returns such a JSON response:

````json

{

"items": [{

"id": "6267f0df-870f-4173-be44-d74b4b56d2bd",

"name": "Human Resources"

}, {

"id": "21c7b61f-330c-489e-8b8c-80e0a78a5cc5",

"name": "Production"

}]

}

````

ABP can now show an auto-complete select component to pick the department while creating or editing a user:

![extension-navigation-property-form](images/extension-navigation-property-form.png)

And shows the department name on the data table:

![extension-navigation-property-form](images/extension-navigation-property-table.png)

#### Lookup Options

`UI.Lookup` has the following options to customize how to read the response returned from the `Url`:

\* `Url`: The endpoint to get the list of target entities. This is used on edit and create forms.

\* `DisplayPropertyName`: The property in the JSON response to read the display name of the target entity to show on the UI. Default: `text`.

\* `ValuePropertyName`: The property in the JSON response to read the Id of the target entity. Default: `id`.

\* `FilterParamName`: ABP allows to search/filter the entity list on edit/create forms. This is especially useful if the target list contains a lot of items. In this case, you can return a limited list (top 100, for example) and allow user to search on the list. ABP sends filter text to the server (as a simple query string) with the name of this option. Default: `filter`.

\* `ResultListPropertyName`: By default, returned JSON result should contain the entity list in an `items` array. You can change the name of this field. Default: `items`.

#### Lookup Properties: How Display Name Works?

You may wonder how ABP shows the department name on the data table above.

It is easy to understand how to fill the dropdown on edit and create forms: ABP makes an AJAX request to the given URL. It re-requests whenever user types to filter the items.

However, for the data table, multiple items are shown on the UI and performing a separate AJAX call to get display name of the department for each row would not be so efficient.

Instead, the display name of the foreign entity is also saved as an extra property of the entity (see *\*Extra Properties\** section of the [Entities](Entities.md) document) in addition to Id of the foreign entity. If you check the database, you can see the `DepartmentId\_Text` in the `ExtraProperties` field in the database table:

````json

{"DepartmentId":"21c7b61f-330c-489e-8b8c-80e0a78a5cc5","DepartmentId\_Text":"Production"}

````

So, this is a type of *\*data duplication\**. If your target entity's name changes in the database later, there is no automatic synchronization system. The system works as expected, but you see the old name on the data tables. If that's a problem for you, you should care yourself to update this information when display name of your entity changes.

## Database Mapping

For relational databases, all extension property values are stored in a single field in the table:

![add-new-propert-to-user-database-extra-properties](images/add-new-propert-to-user-database-extra-properties.png)

`ExtraProperties` field stores the properties as a JSON object. While that's fine for some scenarios, you may want to create a dedicated field for your new property. Fortunately, it is very easy to configure.

If you are using the Entity Framework Core database provider, you can configure the database mapping as shown below:

````csharp

ObjectExtensionManager.Instance

.MapEfCoreProperty<IdentityUser, string>(

"SocialSecurityNumber",

(entityBuilder, propertyBuilder) =>

{

propertyBuilder.HasMaxLength(64);

}

);

````

Write this inside the `YourProjectNameEfCoreEntityExtensionMappings` class in your `.EntityFrameworkCore` project. Then you need to use the standard `Add-Migration` and `Update-Database` commands to create a new database migration and apply the change to your database.

Add-Migration create a new migration as shown below:

````csharp

public partial class Added\_SocialSecurityNumber\_To\_IdentityUser : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.AddColumn<string>(

name: "SocialSecurityNumber",

table: "AbpUsers",

maxLength: 128,

nullable: true);

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropColumn(

name: "SocialSecurityNumber",

table: "AbpUsers");

}

}

````

Once you update your database, you will see that the `AbpUsers` table has the new property as a standard table field:

![add-new-propert-to-user-database-extra-properties](images/add-new-propert-to-user-database-field.png)

> If you first created a property without a database table field, then you later needed to move this property to a database table field, it is suggested to execute an SQL command in your migration to copy the old values to the new field.

>

> However, if you don't make it, the ABP Framework seamlessly manages it. It uses the new database field, but fallbacks to the `ExtraProperties` field if it is null. When you save the entity, it moves the value to the new field.

See the [Extending Entities](Customizing-Application-Modules-Extending-Entities.md) document for more.

## More

See the [Customizing the Modules](Customizing-Application-Modules-Guide.md) guide for an overall index for all the extensibility options.

Here, a few things you can do:

\* You can create a second entity that maps to the same database table with the extra property as a standard class property (if you've defined the EF Core mapping). For the example above, you can add a `public string SocialSecurityNumber {get; set;}` property to the `AppUser` entity in your application, since the `AppUser` entity is mapped to the same `AbpUser` table. Do this only if you need it, since it brings more complexity to your application.

\* You can override a domain or application service to perform custom logics with your new property.

\* You can low level control how to add/render a field in the data table on the UI.

## See Also

\* [Angular UI Extensions](UI/Angular/Extensions-Overall.md)

#### Customizing/Extending Entities

# Customizing the Application Modules: Extending Entities

In some cases, you may want to add some additional properties (and database fields) for an entity defined in a depended module. This section will cover some different approaches to make this possible.

## Extra Properties

[Extra properties](Entities.md) is a way of storing some additional data on an entity without changing it. The entity should implement the `IHasExtraProperties` interface to allow it. All the aggregate root entities defined in the pre-built modules implement the `IHasExtraProperties` interface, so you can store extra properties on these objects.

Example:

````csharp

//SET AN EXTRA PROPERTY

var user = await \_identityUserRepository.GetAsync(userId);

user.SetProperty("Title", "My custom title value!");

await \_identityUserRepository.UpdateAsync(user);

//GET AN EXTRA PROPERTY

var user = await \_identityUserRepository.GetAsync(userId);

return user.GetProperty<string>("Title");

````

This approach is very easy to use and available out of the box. No extra code needed. You can store more than one property at the same time by using different property names (like `Title` here).

Extra properties are stored as a single `JSON` formatted string value in the database for the EF Core. For MongoDB, they are stored as separate fields of the document.

See the [entities document](Entities.md) for more about the extra properties system.

> It is possible to perform a **\*\*business logic\*\*** based on the value of an extra property. You can [override a service method](Customizing-Application-Modules-Overriding-Services.md), then get or set the value as shown above.

## Entity Extensions (EF Core)

As mentioned above, all extra properties of an entity are stored as a single JSON object in the database table. This is not so natural especially when you want to;

\* Create **\*\*indexes\*\*** and **\*\*foreign keys\*\*** for an extra property.

\* Write **\*\*SQL\*\*** or **\*\*LINQ\*\*** using the extra property (search table by the property value, for example).

\* Creating your **\*\*own entity\*\*** maps to the same table, but defines an extra property as a **\*\*regular property\*\*** in the entity (see the [EF Core migration document](Entity-Framework-Core-Migrations.md) for more).

To overcome the difficulties described above, ABP Framework entity extension system for the Entity Framework Core that allows you to use the same extra properties API defined above, but store a desired property as a separate field in the database table.

Assume that you want to add a `SocialSecurityNumber` to the `IdentityUser` entity of the [Identity Module](Modules/Identity.md). You can use the `ObjectExtensionManager`:

````csharp

ObjectExtensionManager.Instance

.MapEfCoreProperty<IdentityUser, string>(

"SocialSecurityNumber",

(entityBuilder, propertyBuilder) =>

{

propertyBuilder.HasMaxLength(32);

}

);

````

\* You provide the `IdentityUser` as the entity name, `string` as the type of the new property, `SocialSecurityNumber` as the property name (also, the field name in the database table).

\* You also need to provide an action that defines the database mapping properties using the [EF Core Fluent API](https://docs.microsoft.com/en-us/ef/core/modeling/entity-properties).

> This code part must be executed before the related `DbContext` used. The [application startup template](Startup-Templates/Application.md) defines a static class named `YourProjectNameEfCoreEntityExtensionMappings`. You can define your extensions in this class to ensure that it is executed in the proper time. Otherwise, you should handle it yourself.

Once you define an entity extension, you then need to use the standard [Add-Migration](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell#add-migration) and [Update-Database](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell#update-database) commands of the EF Core to create a code first migration class and update your database.

You can then use the same extra properties system defined in the previous section to manipulate the property over the entity.

## Creating a New Entity Maps to the Same Database Table/Collection

Another approach can be **\*\*creating your own entity\*\*** mapped to **\*\*the same database table\*\*** (or collection for a MongoDB database).

## Creating a New Entity with Its Own Database Table/Collection

Mapping your entity to an **\*\*existing table\*\*** of a depended module has a few disadvantages;

\* You deal with the **\*\*database migration structure\*\*** for EF Core. While it is possible, you should extra care about the migration code especially when you want to add **\*\*relations\*\*** between entities.

\* Your application database and the module database will be the **\*\*same physical database\*\***. Normally, a module database can be separated if needed, but using the same table restricts it.

If you want to **\*\*loose couple\*\*** your entity with the entity defined by the module, you can create your own database table/collection and map your entity to your own table in your own database.

In this case, you need to deal with the **\*\*synchronization problems\*\***, especially if you want to **\*\*duplicate\*\*** some properties/fields of the related entity. There are a few solutions;

\* If you are building a **\*\*monolithic\*\*** application (or managing your entity and the related module entity within the same process), you can use the [local event bus](Local-Event-Bus.md) to listen changes.

\* If you are building a **\*\*distributed\*\*** system where the module entity is managed (created/updated/deleted) on a different process/service than your entity is managed, then you can subscribe to the [distributed event bus](Distributed-Event-Bus.md) for change events.

Once you handle the event, you can update your own entity in your own database.

### Subscribing to Local Events

[Local Event Bus](Local-Event-Bus.md) system is a way to publish and subscribe to events occurring in the same application.

Assume that you want to get informed when a `IdentityUser` entity changes (created, updated or deleted). You can create a class that implements the `ILocalEventHandler<EntityChangedEventData<IdentityUser>>` interface.

````csharp

public class MyLocalIdentityUserChangeEventHandler :

ILocalEventHandler<EntityChangedEventData<IdentityUser>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityChangedEventData<IdentityUser> eventData)

{

var userId = eventData.Entity.Id;

var userName = eventData.Entity.UserName;

//...

}

}

````

\* `EntityChangedEventData<T>` covers create, update and delete events for the given entity. If you need, you can subscribe to create, update and delete events individually (in the same class or different classes).

\* This code will be executed in the **\*\*current unit of work\*\***, the whole process becomes transactional.

> Reminder: This approach needs to change the `IdentityUser` entity in the same process contains the handler class. It perfectly works even for a clustered environment (when multiple instances of the same application are running on multiple servers).

### Subscribing to Distributed Events

[Distributed Event Bus](Distributed-Event-Bus.md) system is a way to publish an event in one application and receive the event in the same or different application running on the same or different server.

Assume that you want to get informed when `Tenant` entity (of the [Tenant Management](Modules/Tenant-Management.md) module) has created. In this case, you can subscribe to the `EntityCreatedEto<TenantEto>` event as shown in the following example:

````csharp

public class MyDistributedEventHandler :

IDistributedEventHandler<EntityCreatedEto<TenantEto>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityCreatedEto<TenantEto> eventData)

{

var tenantId = eventData.Entity.Id;

var tenantName = eventData.Entity.Name;

//...your custom logic

}

//...

}

````

This handler is executed only when a new tenant has been created. All the pre-built ABP [application modules](Modules/Index.md) define corresponding `ETO` types for their entities. So, you can easily get informed when they changes.

> Notice that ABP doesn't publish distributed events for an entity by default. Because it has a cost and should be enabled by intention. See the [distributed event bus document](Distributed-Event-Bus.md) to learn more.

## See Also

\* [Migration System for the EF Core](Entity-Framework-Core-Migrations.md)

\* [Customizing the Existing Modules](Customizing-Application-Modules-Guide.md)

#### Customizing/Overriding Services

# Customizing the Application Modules: Overriding Services

You may need to **\*\*change behavior (business logic)\*\*** of a depended module for your application. In this case, you can use the power of the [dependency injection system](Dependency-Injection.md) to replace a service, controller or even a page model of the depended module by your own implementation.

**\*\*Replacing a service\*\*** is possible for any type of class registered to the dependency injection, including services of the ABP Framework.

You have different options can be used based on your requirement those will be explained in the next sections.

> Notice that some service methods may not be virtual, so you may not be able to override. We make all virtual by design. If you find any method that is not overridable, please [create an issue](https://github.com/abpframework/abp/issues/new) or do it yourself and send a **\*\*pull request\*\*** on GitHub.

## Replacing an Interface

If given service defines an interface, like the `IdentityUserAppService` class implements the `IIdentityUserAppService`, you can re-implement the same interface and replace the current implementation by your class. Example:

````csharp

public class MyIdentityUserAppService : IIdentityUserAppService, ITransientDependency

{

//...

}

````

`MyIdentityUserAppService` replaces the `IIdentityUserAppService` by naming convention (since both ends with `IdentityUserAppService`). If your class name doesn't match, you need to manually expose the service interface:

````csharp

[ExposeServices(typeof(IIdentityUserAppService))]

public class TestAppService : IIdentityUserAppService, ITransientDependency

{

//...

}

````

The dependency injection system allows to register multiple services for the same interface. The last registered one is used when the interface is injected. It is a good practice to explicitly replace the service.

Example:

````csharp

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IIdentityUserAppService))]

public class TestAppService : IIdentityUserAppService, ITransientDependency

{

//...

}

````

In this way, there will be a single implementation of the `IIdentityUserAppService` interface, while it doesn't change the result for this case. Replacing a service is also possible by code:

````csharp

context.Services.Replace(

ServiceDescriptor.Transient<IIdentityUserAppService, MyIdentityUserAppService>()

);

````

You can write this inside the `ConfigureServices` method of your [module](Module-Development-Basics.md).

## Overriding a Service Class

In most cases, you will want to change one or a few methods of the current implementation for a service. Re-implementing the complete interface would not be efficient in this case. As a better approach, inherit from the original class and override the desired method.

### Example: Overriding an Application Service

````csharp

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IIdentityUserAppService), typeof(IdentityUserAppService), typeof(MyIdentityUserAppService))]

public class MyIdentityUserAppService : IdentityUserAppService

{

//...

public MyIdentityUserAppService(

IdentityUserManager userManager,

IIdentityUserRepository userRepository,

IGuidGenerator guidGenerator

) : base(

userManager,

userRepository,

guidGenerator)

{

}

public async override Task<IdentityUserDto> CreateAsync(IdentityUserCreateDto input)

{

if (input.PhoneNumber.IsNullOrWhiteSpace())

{

throw new AbpValidationException(

"Phone number is required for new users!",

new List<ValidationResult>

{

new ValidationResult(

"Phone number can not be empty!",

new []{"PhoneNumber"}

)

}

); }

return await base.CreateAsync(input);

}

}

````

This class **\*\*overrides\*\*** the `CreateAsync` method of the `IdentityUserAppService` [application service](Application-Services.md) to check the phone number. Then calls the base method to continue to the **\*\*underlying business logic\*\***. In this way, you can perform additional business logic **\*\*before\*\*** and **\*\*after\*\*** the base logic.

You could completely **\*\*re-write\*\*** the entire business logic for a user creation without calling the base method.

### Example: Overriding a Domain Service

````csharp

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IdentityUserManager))]

public class MyIdentityUserManager : IdentityUserManager

{

public MyIdentityUserManager(

IdentityUserStore store,

IIdentityRoleRepository roleRepository,

IIdentityUserRepository userRepository,

IOptions<IdentityOptions> optionsAccessor,

IPasswordHasher<IdentityUser> passwordHasher,

IEnumerable<IUserValidator<IdentityUser>> userValidators,

IEnumerable<IPasswordValidator<IdentityUser>> passwordValidators,

ILookupNormalizer keyNormalizer,

IdentityErrorDescriber errors,

IServiceProvider services,

ILogger<IdentityUserManager> logger,

ICancellationTokenProvider cancellationTokenProvider) :

base(store,

roleRepository,

userRepository,

optionsAccessor,

passwordHasher,

userValidators,

passwordValidators,

keyNormalizer,

errors,

services,

logger,

cancellationTokenProvider)

{

}

public async override Task<IdentityResult> CreateAsync(IdentityUser user)

{

if (user.PhoneNumber.IsNullOrWhiteSpace())

{

throw new AbpValidationException(

"Phone number is required for new users!",

new List<ValidationResult>

{

new ValidationResult(

"Phone number can not be empty!",

new []{"PhoneNumber"}

)

}

);

}

return await base.CreateAsync(user);

}

}

````

This example class inherits from the `IdentityUserManager` [domain service](Domain-Services.md) and overrides the `CreateAsync` method to perform the same phone number check implemented above. The result is same, but this time we've implemented it inside the domain service assuming that this is a **\*\*core domain logic\*\*** for our system.

> `[ExposeServices(typeof(IdentityUserManager))]` attribute is **\*\*required\*\*** here since `IdentityUserManager` does not define an interface (like `IIdentityUserManager`) and dependency injection system doesn't expose services for inherited classes (like it does for the implemented interfaces) by convention.

Check the [localization system](Localization.md) to learn how to localize the error messages.

### Example: Overriding a Repository

````csharp

public class MyEfCoreIdentityUserRepository : EfCoreIdentityUserRepository

{

public MyEfCoreIdentityUserRepository(

IDbContextProvider<IIdentityDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

/\* You can override any base method here \*/

}

````

In this example, we are overriding the `EfCoreIdentityUserRepository` class that is defined by the [Identity module](Modules/Identity.md). This is the [Entity Framework Core](Entity-Framework-Core.md) implementation of the user repository.

Thanks to the naming convention (`MyEfCoreIdentityUserRepository` ends with `EfCoreIdentityUserRepository`), no additional setup is required. You can override any base method to customize it for your needs.

However, if you inject `IRepository<IdentityUser>` or `IRepository<IdentityUser, Guid>`, it will still use the default repository implementation. To replace the default repository implementation, write the following code in the `ConfigureServices` method of your module class:

````csharp

context.Services.AddDefaultRepository(

typeof(Volo.Abp.Identity.IdentityUser),

typeof(MyEfCoreIdentityUserRepository),

replaceExisting: true

);

````

In this way, your implementation will be used if you inject `IRepository<IdentityUser>`, `IRepository<IdentityUser, Guid>` or `IIdentityUserRepository`.

If you want to add extra methods to your repository and use it in your own code, you can define an interface and expose it from your repository implementation. You can also extend the pre-built repository interface. Example:

````csharp

public interface IMyIdentityUserRepository : IIdentityUserRepository

{

public Task DeleteByEmailAddress(string email);

}

````

The `IMyIdentityUserRepository` interface extends the Identity module's `IIdentityUserRepository` interface. Then you can implement it as shown in the following example:

````csharp

[ExposeServices(typeof(IMyIdentityUserRepository), IncludeDefaults = true)]

public class MyEfCoreIdentityUserRepository

: EfCoreIdentityUserRepository, IMyIdentityUserRepository

{

public MyEfCoreIdentityUserRepository(

IDbContextProvider<IIdentityDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task DeleteByEmailAddress(string email)

{

var dbContext = await GetDbContextAsync();

var user = await dbContext.Users.FirstOrDefaultAsync(u => u.Email == email);

if (user != null)

{

dbContext.Users.Remove(user);

}

}

}

````

The `MyEfCoreIdentityUserRepository` class implements the `IMyIdentityUserRepository` interface. `ExposeServices` attribute is needed since ABP can not expose `IMyIdentityUserRepository` by naming conventions (`MyEfCoreIdentityUserRepository` doesn't end with `MyIdentityUserRepository`). Now, you can inject the `IMyIdentityUserRepository` interface into your services and call its `DeleteByEmailAddress` method.

### Example: Overriding a Controller

````csharp

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

using Volo.Abp.Account;

using Volo.Abp.DependencyInjection;

namespace MyProject.Controllers

{

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(AccountController))]

public class MyAccountController : AccountController

{

public MyAccountController(IAccountAppService accountAppService)

: base(accountAppService)

{

}

public async override Task SendPasswordResetCodeAsync(

SendPasswordResetCodeDto input)

{

Logger.LogInformation("Your custom logic...");

await base.SendPasswordResetCodeAsync(input);

}

}

}

````

This example replaces the `AccountController` (An API Controller defined in the [Account Module](Modules/Account.md)) and overrides the `SendPasswordResetCodeAsync` method.

**\*\***`[ExposeServices(typeof(AccountController))]` **is essential\*\*** here since it registers this controller for the `AccountController` in the dependency injection system. `[Dependency(ReplaceServices = true)]` is also recommended to clear the old registration (even the ASP.NET Core DI system selects the last registered one).

In addition, the `MyAccountController` will be removed from [`ApplicationModel`](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.applicationmodels.applicationmodel.controllers) because it defines `ExposeServicesAttribute`.

If `IncludeSelf = true` is specified, i.e. `[ExposeServices(typeof(AccountController), IncludeSelf = true)]`, then `AccountController` will be removed instead. This is useful for **\*\*extending\*\*** a controller.

If you don't want to remove either controller, you can configure `AbpAspNetCoreMvcOptions`:

```csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.IgnoredControllersOnModelExclusion

.AddIfNotContains(typeof(MyAccountController));

});

```

### Overriding Other Classes

Overriding controllers, framework services, view component classes and any other type of classes registered to dependency injection can be overridden just like the examples above.

## Extending Data Transfer Objects

**\*\*Extending [**entities**](Entities.md)\*\*** is possible as described in the [Extending Entities document](Customizing-Application-Modules-Extending-Entities.md). In this way, you can add **\*\*custom properties\*\*** to entities and perform **\*\*additional business logic\*\*** by overriding the related services as described above.

It is also possible to extend Data Transfer Objects (**\*\*DTOs\*\***) used by the application services. In this way, you can get extra properties from the UI (or client) and return extra properties from the service.

### Example

Assuming that you've already added a `SocialSecurityNumber` as described in the [Extending Entities document](Customizing-Application-Modules-Extending-Entities.md) and want to include this information while getting the list of users from the `GetListAsync` method of the `IdentityUserAppService`.

You can use the [object extension system](Object-Extensions.md) to add the property to the `IdentityUserDto`. Write this code inside the `YourProjectNameDtoExtensions` class comes with the application startup template:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUserDto, string>(

"SocialSecurityNumber"

);

````

This code defines a `SocialSecurityNumber` to the `IdentityUserDto` class as a `string` type. That's all. Now, if you call the `/api/identity/users` HTTP API (which uses the `IdentityUserAppService` internally) from a REST API client, you will see the `SocialSecurityNumber` value in the `extraProperties` section.

````json

{

"totalCount": 1,

"items": [{

"tenantId": null,

"userName": "admin",

"name": "admin",

"surname": null,

"email": "admin@abp.io",

"emailConfirmed": false,

"phoneNumber": null,

"phoneNumberConfirmed": false,

"twoFactorEnabled": false,

"lockoutEnabled": true,

"lockoutEnd": null,

"concurrencyStamp": "b4c371a0ab604de28af472fa79c3b70c",

"isDeleted": false,

"deleterId": null,

"deletionTime": null,

"lastModificationTime": "2020-04-09T21:25:47.0740706",

"lastModifierId": null,

"creationTime": "2020-04-09T21:25:46.8308744",

"creatorId": null,

"id": "8edecb8f-1894-a9b1-833b-39f4725db2a3",

"extraProperties": {

"SocialSecurityNumber": "123456789"

}

}]

}

````

Manually added the `123456789` value to the database for now.

All pre-built modules support extra properties in their DTOs, so you can configure easily.

### Definition Check

When you [define](Customizing-Application-Modules-Extending-Entities.md) an extra property for an entity, it doesn't automatically appear in all the related DTOs, because of the security. The extra property may contain a sensitive data and you may not want to expose it to the clients by default.

So, you need to explicitly define the same property for the corresponding DTO if you want to make it available for the DTO (as just done above). If you want to allow to set it on user creation, you also need to define it for the `IdentityUserCreateDto`.

If the property is not so secure, this can be tedious. Object extension system allows you to ignore this definition check for a desired property. See the example below:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.MapEfCore(b => b.HasMaxLength(32));

options.CheckPairDefinitionOnMapping = false;

}

);

````

This is another approach to define a property for an entity (`ObjectExtensionManager` has more, see [its document](Object-Extensions.md)). This time, we set `CheckPairDefinitionOnMapping` to false to skip definition check while mapping entities to DTOs and vice verse.

If you don't like this approach but want to add a single property to multiple objects (DTOs) easier, `AddOrUpdateProperty` can get an array of types to add the extra property:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<string>(

new[]

{

typeof(IdentityUserDto),

typeof(IdentityUserCreateDto),

typeof(IdentityUserUpdateDto)

},

"SocialSecurityNumber"

);

````

### About the User Interface

This system allows you to add extra properties to entities and DTOs and execute custom business code, however it does nothing related to the User Interface.

See [Overriding the User Interface](Customizing-Application-Modules-Overriding-User-Interface.md) guide for the UI part.

## How to Find the Services?

[Module documents](Modules/Index.md) includes the list of the major services they define. In addition, you can investigate [their source code](https://github.com/abpframework/abp/tree/dev/modules) to explore all the services.

## Domain Driven Design

### Overall

# Domain Driven Design

## What is DDD?

ABP framework provides an **\*\*infrastructure\*\*** to make **\*\*Domain Driven Design\*\*** based development easier to implement. DDD is [defined in the Wikipedia](https://en.wikipedia.org/wiki/Domain-driven\_design) as below:

> **\*\*Domain-driven design\*\*** (**\*\*DDD\*\***) is an approach to software development for complex needs by connecting the implementation to an evolving model. The premise of domain-driven design is the following:

>

> - Placing the project's primary focus on the core domain and domain logic;

> - Basing complex designs on a model of the domain;

> - Initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

## Layers & Building Blocks

ABP follows DDD principles and patterns to achieve a layered application model which consists of four fundamental layers:

- **\*\*Presentation Layer\*\***: Provides an interface to the user. Uses the *\*Application Layer\** to achieve user interactions.

- **\*\*Application Layer\*\***: Mediates between the Presentation and Domain Layers. Orchestrates business objects to perform specific application tasks. Implements use cases as the application logic.

- **\*\*Domain Layer\*\***: Includes business objects and the core (domain) business rules. This is the heart of the application.

- **\*\*Infrastructure Layer\*\***: Provides generic technical capabilities that support higher layers mostly using 3rd-party libraries.

DDD mostly interest in the **\*\*Domain\*\*** and the **\*\*Application\*\*** layers, rather than the Infrastructure and the Presentation layers. The following documents explains the **\*\*infrastructure\*\*** provided by the ABP Framework to implement **\*\*Building Blocks\*\*** of the DDD:

\* **\*\*Domain Layer\*\***

\* [Entities & Aggregate Roots](Entities.md)

\* [Repositories](Repositories.md)

\* [Domain Services](Domain-Services.md)

\* [Value Objects](Value-Objects.md)

\* [Specifications](Specifications.md)

\* **\*\*Application Layer\*\***

\* [Application Services](Application-Services.md)

\* [Data Transfer Objects (DTOs)](Data-Transfer-Objects.md)

\* [Unit of Work](Unit-Of-Work.md)

## Free E-Book: Implementing DDD

See the [Implementing Domain Driven Design book](https://abp.io/books/implementing-domain-driven-design) as a **\*\*complete reference\*\***. This book explains the Domain Driven Design and introduces explicit **\*\*rules and examples\*\*** to give a deep understanding of the **\*\*implementation details\*\***.

### Domain Layer

#### Entities & Aggregate Roots

# Entities

Entities are one of the core concepts of DDD (Domain Driven Design). Eric Evans describes it as "*\*An object that is not fundamentally defined by its attributes, but rather by a thread of continuity and identity\**".

An entity is generally mapped to a table in a relational database.

## Entity Class

Entities are derived from the `Entity<TKey>` class as shown below:

```C#

public class Book : Entity<Guid>

{

public string Name { get; set; }

public float Price { get; set; }

}

```

> If you do not want to derive your entity from the base `Entity<TKey>` class, you can directly implement `IEntity<TKey>` interface.

`Entity<TKey>` class just defines an `Id` property with the given primary **\*\*key type\*\***, which is `Guid` in the example above. It can be other types like `string`, `int`, `long`, or whatever you need.

### Entities with GUID Keys

If your entity's Id type is `Guid`, there are some good practices to implement:

\* Create a constructor that gets the Id as a parameter and passes to the base class.

\* If you don't set a GUID Id, **\*\*ABP Framework sets it on save\*\***, but it is good to have a valid Id on the entity even before saving it to the database.

\* If you create an entity with a constructor that takes parameters, also create a `private` or `protected` empty constructor. This is used while your database provider reads your entity from the database (on deserialization).

\* Don't use the `Guid.NewGuid()` to set the Id! **\*\*Use [**the `IGuidGenerator` service**](Guid-Generation.md)\*\*** while passing the Id from the code that creates the entity. `IGuidGenerator` optimized to generate sequential GUIDs, which is critical for clustered indexes in the relational databases.

An example entity:

````csharp

public class Book : Entity<Guid>

{

public string Name { get; set; }

public float Price { get; set; }

protected Book()

{

}

public Book(Guid id)

: base(id)

{

}

}

````

Example usage in an [application service](Application-Services.md):

````csharp

public class BookAppService : ApplicationService, IBookAppService

{

private readonly IRepository<Book> \_bookRepository;

public BookAppService(IRepository<Book> bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task CreateAsync(CreateBookDto input)

{

await \_bookRepository.InsertAsync(

new Book(GuidGenerator.Create())

{

Name = input.Name,

Price = input.Price

}

);

}

}

````

\* `BookAppService` injects the default [repository](Repositories.md) for the book entity and uses its `InsertAsync` method to insert a `Book` to the database.

\* `GuidGenerator` is type of `IGuidGenerator` which is a property defined in the `ApplicationService` base class. ABP defines such frequently used base properties as pre-injected for you, so you don't need to manually [inject](Dependency-Injection.md) them.

\* If you want to follow the DDD best practices, see the *\*Aggregate Example\** section below.

### Entities with Composite Keys

Some entities may need to have **\*\*composite keys\*\***. In that case, you can derive your entity from the non-generic `Entity` class. Example:

````C#

public class UserRole : Entity

{

public Guid UserId { get; set; }

public Guid RoleId { get; set; }

public DateTime CreationTime { get; set; }

public UserRole()

{

}

public override object[] GetKeys()

{

return new object[] { UserId, RoleId };

}

}

````

For the example above, the composite key is composed of `UserId` and `RoleId`. For a relational database, it is the composite primary key of the related table. Entities with composite keys should implement the `GetKeys()` method as shown above.

> Notice that you also need to define keys of the entity in your **\*\*object-relational mapping\*\*** (ORM) configuration. See the [Entity Framework Core](Entity-Framework-Core.md) integration document for example.

> Also note that Entities with Composite Primary Keys cannot utilize the `IRepository<TEntity, TKey>` interface since it requires a single Id property. However, you can always use `IRepository<TEntity>`. See [repositories documentation](Repositories.md) for more.

### EntityEquals

`Entity.EntityEquals(...)` method is used to check if two Entity Objects are equals.

Example:

```csharp

Book book1 = ...

Book book2 = ...

if (book1.EntityEquals(book2)) //Check equality

{

...

}

```

## AggregateRoot Class

"*\*Aggregate is a pattern in Domain-Driven Design. A DDD aggregate is a cluster of domain objects that can be treated as a single unit. An example may be an order and its line-items, these will be separate objects, but it's useful to treat the order (together with its line items) as a single aggregate.\**" (see the [full description](http://martinfowler.com/bliki/DDD\_Aggregate.html))

`AggregateRoot<TKey>` class extends the `Entity<TKey>` class. So, it also has an `Id` property by default.

> Notice that ABP creates default repositories only for aggregate roots by default. However, it's possible to include all entities. See the [repositories documentation](Repositories.md) for more.

ABP does not force you to use aggregate roots, you can in fact use the `Entity` class as defined before. However, if you want to implement the [Domain Driven Design](Domain-Driven-Design.md) and want to create aggregate root classes, there are some best practices you may want to consider:

\* An aggregate root is responsible to preserve it's own integrity. This is also true for all entities, but aggregate root has responsibility for it's sub entities too. So, the aggregate root must always be in a valid state.

\* An aggregate root can be referenced by it's Id. Do not reference it by it's navigation property.

\* An aggregate root is treated as a single unit. It's retrieved and updated as a single unit. It's generally considered as a transaction boundary.

\* Work with sub-entities over the aggregate root- do not modify them independently.

See the [entity design best practice guide](Best-Practices/Entities.md) if you want to implement DDD in your application.

### Aggregate Example

This is a full sample of an aggregate root with a related sub-entity collection:

````C#

public class Order : AggregateRoot<Guid>

{

public virtual string ReferenceNo { get; protected set; }

public virtual int TotalItemCount { get; protected set; }

public virtual DateTime CreationTime { get; protected set; }

public virtual List<OrderLine> OrderLines { get; protected set; }

protected Order()

{

}

public Order(Guid id, string referenceNo)

{

Check.NotNull(referenceNo, nameof(referenceNo));

Id = id;

ReferenceNo = referenceNo;

OrderLines = new List<OrderLine>();

}

public void AddProduct(Guid productId, int count)

{

if (count <= 0)

{

throw new ArgumentException(

"You can not add zero or negative count of products!",

nameof(count)

);

}

var existingLine = OrderLines.FirstOrDefault(ol => ol.ProductId == productId);

if (existingLine == null)

{

OrderLines.Add(new OrderLine(this.Id, productId, count));

}

else

{

existingLine.ChangeCount(existingLine.Count + count);

}

TotalItemCount += count;

}

}

public class OrderLine : Entity

{

public virtual Guid OrderId { get; protected set; }

public virtual Guid ProductId { get; protected set; }

public virtual int Count { get; protected set; }

protected OrderLine()

{

}

internal OrderLine(Guid orderId, Guid productId, int count)

{

OrderId = orderId;

ProductId = productId;

Count = count;

}

internal void ChangeCount(int newCount)

{

Count = newCount;

}

public override object[] GetKeys()

{

return new Object[] {OrderId, ProductId};

}

}

````

> If you do not want to derive your aggregate root from the base `AggregateRoot<TKey>` class, you can directly implement the `IAggregateRoot<TKey>` interface.

`Order` is an **\*\*aggregate root\*\*** with `Guid` type `Id` property. It has a collection of `OrderLine` entities. `OrderLine` is another entity with a composite primary key (`OrderId` and ` ProductId`).

While this example may not implement all the best practices of an aggregate root, it still follows some good practices:

\* `Order` has a public constructor that takes **\*\*minimal requirements\*\*** to construct an `Order` instance. So, it's not possible to create an order without an id and reference number. The **\*\*protected/private\*\*** constructor is only necessary to **\*\*deserialize\*\*** the object while reading from a data source.

\* `OrderLine` constructor is internal, so it is only allowed to be created by the domain layer. It's used inside of the `Order.AddProduct` method.

\* `Order.AddProduct` implements the business rule to add a product to an order.

\* All properties have `protected` setters. This is to prevent the entity from arbitrary changes from outside of the entity. For example, it would be dangerous to set `TotalItemCount` without adding a new product to the order. It's value is maintained by the `AddProduct` method.

ABP Framework does not force you to apply any DDD rule or patterns. However, it tries to make it possible and easier when you do want to apply them. The documentation also follows the same principle.

### Aggregate Roots with Composite Keys

While it's not common (and not suggested) for aggregate roots, it is in fact possible to define composite keys in the same way as defined for the mentioned entities above. Use non-generic `AggregateRoot` base class in that case.

### BasicAggregateRoot Class

`AggregateRoot` class implements the `IHasExtraProperties` and `IHasConcurrencyStamp` interfaces which brings two properties to the derived class. `IHasExtraProperties` makes the entity extensible (see the *\*Extra Properties\** section below) and `IHasConcurrencyStamp` adds a `ConcurrencyStamp` property that is managed by the ABP Framework to implement the [optimistic concurrency](https://docs.microsoft.com/en-us/ef/core/saving/concurrency). In most cases, these are wanted features for aggregate roots.

However, if you don't need these features, you can inherit from the `BasicAggregateRoot<TKey>` (or `BasicAggregateRoot`) for your aggregate root.

## Base Classes & Interfaces for Audit Properties

There are some properties like `CreationTime`, `CreatorId`, `LastModificationTime`... which are very common in all applications. ABP Framework provides some interfaces and base classes to **\*\*standardize\*\*** these properties and also **\*\*sets their values automatically\*\***.

### Auditing Interfaces

There are a lot of auditing interfaces, so you can implement the one that you need.

> While you can manually implement these interfaces, you can use **\*\*the base classes\*\*** defined in the next section to simplify it.

\* `IHasCreationTime` defines the following properties:

\* `CreationTime`

\* `IMayHaveCreator` defines the following properties:

\* `CreatorId`

\* `ICreationAuditedObject` inherits from the `IHasCreationTime` and the `IMayHaveCreator`, so it defines the following properties:

\* `CreationTime`

\* `CreatorId`

\* `IHasModificationTime` defines the following properties:

\* `LastModificationTime`

\* `IModificationAuditedObject` extends the `IHasModificationTime` and adds the `LastModifierId` property. So, it defines the following properties:

\* `LastModificationTime`

\* `LastModifierId`

\* `IAuditedObject` extends the `ICreationAuditedObject` and the `IModificationAuditedObject`, so it defines the following properties:

\* `CreationTime`

\* `CreatorId`

\* `LastModificationTime`

\* `LastModifierId`

\* `ISoftDelete` (see the [data filtering document](Data-Filtering.md)) defines the following properties:

\* `IsDeleted`

\* `IHasDeletionTime` extends the `ISoftDelete` and adds the `DeletionTime` property. So, it defines the following properties:

\* `IsDeleted`

\* `DeletionTime`

\* `IDeletionAuditedObject` extends the `IHasDeletionTime` and adds the `DeleterId` property. So, it defines the following properties:

\* `IsDeleted`

\* `DeletionTime`

\* `DeleterId`

\* `IFullAuditedObject` inherits from the `IAuditedObject` and the `IDeletionAuditedObject`, so it defines the following properties:

\* `CreationTime`

\* `CreatorId`

\* `LastModificationTime`

\* `LastModifierId`

\* `IsDeleted`

\* `DeletionTime`

\* `DeleterId`

Once you implement any of the interfaces, or derive from a class defined in the next section, ABP Framework automatically manages these properties wherever possible.

> Implementing `ISoftDelete`, `IDeletionAuditedObject` or `IFullAuditedObject` makes your entity **\*\*soft-delete\*\***. See the [data filtering document](Data-Filtering.md) to learn about the soft-delete pattern.

### Auditing Base Classes

While you can manually implement any of the interfaces defined above, it is suggested to inherit from the base classes defined here:

\* `CreationAuditedEntity<TKey>` and `CreationAuditedAggregateRoot<TKey>` implement the `ICreationAuditedObject` interface.

\* `AuditedEntity<TKey>` and `AuditedAggregateRoot<TKey>` implement the `IAuditedObject` interface.

\* `FullAuditedEntity<TKey>` and `FullAuditedAggregateRoot<TKey>` implement the `IFullAuditedObject` interface.

All these base classes also have non-generic versions to take `AuditedEntity` and `FullAuditedAggregateRoot` to support the composite primary keys.

All these base classes also have `...WithUser` pairs, like `FullAuditedAggregateRootWithUser<TUser>` and `FullAuditedAggregateRootWithUser<TKey, TUser>`. This makes possible to add a navigation property to your user entity. However, it is not a good practice to add navigation properties between aggregate roots, so this usage is not suggested (unless you are using an ORM, like EF Core, that well supports this scenario and you really need it - otherwise remember that this approach doesn't work for NoSQL databases like MongoDB where you must truly implement the aggregate pattern). Also, if you add navigation properties to the AppUser class that comes with the startup template, consider to handle (ignore/map) it on the migration dbcontext (see [the EF Core migration document](Entity-Framework-Core-Migrations.md)).

## Caching Entities

ABP Framework provides a [Distributed Entity Cache System](Entity-Cache.md) for caching entities. It is useful if you want to use caching for quicker access to the entity rather than repeatedly querying it from the database.

It's designed as read-only and automatically invalidates a cached entity if the entity is updated or deleted.

> See the [Entity Cache](Entity-Cache.md) documentation for more information.

## Versioning Entities

ABP defines the `IHasEntityVersion` interface for automatic versioning of your entities. It only provides a single `EntityVersion` property, as shown in the following code block:

````csharp

public interface IHasEntityVersion

{

int EntityVersion { get; }

}

````

If you implement the `IHasEntityVersion` interface, ABP automatically increases the `EntityVersion` value whenever you update your entity. The initial `EntityVersion` value will be `0`, when you first create an entity and save to the database.

> ABP can not increase the version if you directly execute SQL `UPDATE` commands in the database. It is your responsibility to increase the `EntityVersion` value in that case. Also, if you are using the aggregate pattern and change sub-collections of an aggregate root, it is your responsibility if you want to increase the version of the aggregate root object.

## Extra Properties

ABP defines the `IHasExtraProperties` interface that can be implemented by an entity to be able to dynamically set and get properties for the entity. `AggregateRoot` base class already implements the `IHasExtraProperties` interface. If you've derived from this class (or one of the related audit class defined above), you can directly use the API.

### GetProperty & SetProperty Extension Methods

These extension methods are the recommended way to get and set data for an entity. Example:

````csharp

public class ExtraPropertiesDemoService : ITransientDependency

{

private readonly IIdentityUserRepository \_identityUserRepository;

public ExtraPropertiesDemoService(IIdentityUserRepository identityUserRepository)

{

\_identityUserRepository = identityUserRepository;

}

public async Task SetTitle(Guid userId, string title)

{

var user = await \_identityUserRepository.GetAsync(userId);

//SET A PROPERTY

user.SetProperty("Title", title);

await \_identityUserRepository.UpdateAsync(user);

}

public async Task<string> GetTitle(Guid userId)

{

var user = await \_identityUserRepository.GetAsync(userId);

//GET A PROPERTY

return user.GetProperty<string>("Title");

}

}

````

\* Property's **\*\*value is object\*\*** and can be any type of object (string, int, bool... etc).

\* `GetProperty` returns `null` if given property was not set before.

\* You can store more than one property at the same time by using different **\*\*property names\*\*** (like `Title` here).

It would be a good practice to **\*\*define a constant\*\*** for the property name to prevent typo errors. It would be even a better practice to **\*\*define extension methods\*\*** to take the advantage of the intellisense. Example:

````csharp

public static class IdentityUserExtensions

{

private const string TitlePropertyName = "Title";

public static void SetTitle(this IdentityUser user, string title)

{

user.SetProperty(TitlePropertyName, title);

}

public static string GetTitle(this IdentityUser user)

{

return user.GetProperty<string>(TitlePropertyName);

}

}

````

Then you can directly use `user.SetTitle("...")` and `user.GetTitle()` for an `IdentityUser` object.

### HasProperty & RemoveProperty Extension Methods

\* `HasProperty` is used to check if the object has a property set before.

\* `RemoveProperty` is used to remove a property from the object. You can use this instead of setting a `null` value.

### How it is Implemented?

`IHasExtraProperties` interface requires to define a `Dictionary<string, object>` property, named `ExtraProperties`, for the implemented class.

So, you can directly use the `ExtraProperties` property to use the dictionary API, if you like. However, `SetProperty` and `GetProperty` methods are the recommended ways since they also check for `null`s.

#### How is it Stored?

The way to store this dictionary in the database depends on the database provider you're using.

\* For [Entity Framework Core](Entity-Framework-Core.md), here are two type of configurations;

\* By default, it is stored in a single `ExtraProperties` field as a `JSON` string (that means all extra properties stored in a single database table field). Serializing to `JSON` and deserializing from the `JSON` are automatically done by the ABP Framework using the [value conversions](https://docs.microsoft.com/en-us/ef/core/modeling/value-conversions) system of the EF Core.

\* If you want, you can use the `ObjectExtensionManager` to define a separate table field for a desired extra property. Properties those are not configured through the `ObjectExtensionManager` will continue to use a single `JSON` field as described above. This feature is especially useful when you are using a pre-built [application module](Modules/Index.md) and want to [extend its entities](Customizing-Application-Modules-Extending-Entities.md). See the [EF Core integration document](Entity-Framework-Core.md) to learn how to use the `ObjectExtensionManager`.

\* For [MongoDB](MongoDB.md), it is stored as a **\*\*regular field\*\***, since MongoDB naturally supports this kind of [extra elements](https://mongodb.github.io/mongo-csharp-driver/1.11/serialization/#supporting-extra-elements) system.

### Discussion for the Extra Properties

Extra Properties system is especially useful if you are using a **\*\*re-usable module\*\*** that defines an entity inside and you want to get/set some data related to this entity in an easy way.

You typically **\*\*don't need\*\*** to use this system for your own entities, because it has the following drawbacks:

\* It is **\*\*not fully type safe\*\*** since it works with strings as property names.

\* It is **\*\*not easy to [**auto map**](Object-To-Object-Mapping.md)\*\*** these properties from/to other objects.

### Extra Properties Behind Entities

`IHasExtraProperties` is not restricted to be used with entities. You can implement this interface for any kind of class and use the `GetProperty`, `SetProperty` and other related methods.

## See Also

\* [Best practice guide to design the entities](Best-Practices/Entities.md)

#### Value Objects

# Value Objects

> An object that represents a descriptive aspect of the domain with no conceptual identity is called a VALUE OBJECT.

>

> (Eric Evans)

Two [Entities](Entities.md) with the same properties but with different `Id`s are considered as different entities. However, Value Objects have no `Id`s and they are considered as equals if they have the same property values.

## The ValueObject Class

`ValueObject` is an abstract class that can be inherited to create a Value Object class.

**\*\*Example: An Address class\*\***

````csharp

public class Address : ValueObject

{

public Guid CityId { get; private set; }

public string Street { get; private set; }

public int Number { get; private set; }

private Address()

{

}

public Address(

Guid cityId,

string street,

int number)

{

CityId = cityId;

Street = street;

Number = number;

}

protected override IEnumerable<object> GetAtomicValues()

{

yield return Street;

yield return CityId;

yield return Number;

}

}

````

\* A Value Object class must implement the `GetAtomicValues()` method to return the primitive values.

### ValueEquals

`ValueObject.ValueEquals(...)` method is used to check if two Value Objects are equals.

**\*\*Example: Check if two addresses are equals\*\***

````csharp

Address address1 = ...

Address address2 = ...

if (address1.ValueEquals(address2)) //Check equality

{

...

}

````

## Best Practices

Here are some best practices when using Value Objects:

- Design a value object as **\*\*immutable\*\*** (like the Address above) if there is not a good reason for designing it as mutable.

- The properties that make up a Value Object should form a conceptual whole. For example, CityId, Street and Number shouldn't be separate properties of a Person entity. This also makes the Person entity simpler.

## See Also

\* [Entities](Entities.md)

#### Repositories

# Repositories

"*\*Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects\**" (Martin Fowler).

Repositories, in practice, are used to perform database operations for domain objects (see [Entities](Entities.md)). Generally, a separated repository is used for each **\*\*aggregate root\*\*** or entity.

## Generic Repositories

ABP can provide a **\*\*default generic repository\*\*** for each aggregate root or entity. You can [inject](Dependency-Injection.md) `IRepository<TEntity, TKey>` into your service and perform standard **\*\*CRUD\*\*** operations.

> Database provider layer should be properly configured to be able to use the default generic repositories. It is **\*\*already done\*\*** if you've created your project using the startup templates. If not, refer to the database provider documents ([EF Core](Entity-Framework-Core.md) / [MongoDB](MongoDB.md)) to configure it.

**\*\*Example usage of a default generic repository:\*\***

````C#

using System;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Demo

{

public class PersonAppService : ApplicationService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task CreateAsync(CreatePersonDto input)

{

var person = new Person(input.Name);

await \_personRepository.InsertAsync(person);

}

public async Task<int> GetCountAsync(string filter)

{

return await \_personRepository.CountAsync(p => p.Name.Contains(filter));

}

}

}

````

In this example;

\* `PersonAppService` simply injects `IRepository<Person, Guid>` in it's constructor.

\* `CreateAsync` method uses `InsertAsync` to save the new entity.

\* `GetCountAsync` method gets a filtered count of all people in the database.

### Standard Repository Methods

Generic Repositories provides some standard CRUD features out of the box:

\* `GetAsync`: Returns a single entity by its `Id` or a predicate (lambda expression).

\* Throws `EntityNotFoundException` if the requested entity was not found.

\* Throws `InvalidOperationException` if there are multiple entities with given predicate.

\* `FindAsync`: Returns a single entity by its `Id` or a predicate (lambda expression).

\* Returns `null` if the requested entity was not found.

\* Throws `InvalidOperationException` if there are multiple entities with given predicate.

\* `InsertAsync`: Inserts a new entity to the database.

\* `UpdateAsync`: Updates an existing entity in the database.

\* `DeleteAsync`: Deletes the given entity from database.

\* This method has an overload that takes a predicate (lambda expression) to delete multiple entities satisfies the given condition.

\* `GetListAsync`: Returns the list of all entities in the database.

\* `GetPagedListAsync`: Returns a limited list of entities. Gets `skipCount`, `maxResultCount` and `sorting` parameters.

\* `GetCountAsync`: Gets count of all entities in the database.

There are overloads of these methods.

\* Provides `UpdateAsync` and `DeleteAsync` methods to update or delete an entity by entity object or it's id.

\* Provides `DeleteAsync` method to delete multiple entities by a filter.

### Querying / LINQ over the Repositories

Repositories provide the `GetQueryableAsync()` method that returns an `IQueryable<TEntity>` object. You can use this object to perform LINQ queries on the entities in the database.

**\*\*Example: Use LINQ with the repositories\*\***

````csharp

using System;

using System.Linq;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Demo

{

public class PersonAppService : ApplicationService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task<List<PersonDto>> GetListAsync(string filter)

{

//Obtain the IQueryable<Person>

IQueryable<Person> queryable = await \_personRepository.GetQueryableAsync();

//Create a query

var query = from person in queryable

where person.Name == filter

orderby person.Name

select person;

//Execute the query to get list of people

var people = query.ToList();

//Convert to DTO and return to the client

return people.Select(p => new PersonDto {Name = p.Name}).ToList();

}

}

}

````

You could also use the LINQ extension methods:

````csharp

public async Task<List<PersonDto>> GetListAsync(string filter)

{

//Obtain the IQueryable<Person>

IQueryable<Person> queryable = await \_personRepository.GetQueryableAsync();

//Execute a query

var people = queryable

.Where(p => p.Name.Contains(filter))

.OrderBy(p => p.Name)

.ToList();

//Convert to DTO and return to the client

return people.Select(p => new PersonDto {Name = p.Name}).ToList();

}

````

Any standard LINQ method can be used over the `IQueryable` returned from the repository.

> This sample uses `ToList()` method, but it is **\*\*strongly suggested to use the asynchronous methods\*\*** to perform database queries, like `ToListAsync()` for this example. See the **\*\***`IQueryable` **& Async Operations\*\*** section to learn how you can do it.

> **\*\*Exposing** `IQueryable` **to outside of a repository\*\*** class may leak your data access logic to the application layer. If you want to strictly follow the **\*\*layered architecture\*\*** principles, you can consider to implement a custom repository class and wrap your data access logic inside your repository class. You can see the **\*\**\*Custom Repositories\*\*****\** section to learn how to create custom repository classes for your application.

### Bulk Operations

There are some methods to perform bulk operations in the database;

\* `InsertManyAsync`

\* `UpdateManyAsync`

\* `DeleteManyAsync`

These methods work with multiple entities and can take advantage of bulk operations if supported by the underlying database provider.

> Optimistic concurrency control may not be possible when you use `UpdateManyAsync` and `DeleteManyAsync` methods.

### Soft / Hard Delete

`DeleteAsync` method of the repository doesn't delete the entity if the entity is a **\*\*soft-delete\*\*** entity (that implements `ISoftDelete`). Soft-delete entities are marked as "deleted" in the database. Data Filter system ensures that the soft deleted entities are not retrieved from database normally.

If your entity is a soft-delete entity, you can use the `HardDeleteAsync` method to physically delete the entity from database in case of you need it.

> See the [Data Filtering](Data-Filtering.md) documentation for more about soft-delete.

### Delete Direct

`DeleteDirectAsync` method of the repository deletes all entities those fit to the given predicate. It directly deletes entities from database, without fetching them.

Some features (like soft-delete, multi-tenancy and audit logging) won't work, so use this method carefully when you need it. Use the `DeleteAsync` method if you need to these features.

> Currently only [EF Core supports it](https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-7.0/whatsnew#basic-executedelete-examples), For the ORMs doesn't support direct delete, we will fallback to the existing `DeleteAsync` method.

### Ensure Entities Exists

The `EnsureExistsAsync` extension method accepts entity id or entities query expression to ensure entities exist, otherwise, it will throw `EntityNotFoundException`.

## Other Generic Repository Types

Standard `IRepository<TEntity, TKey>` interface exposes the standard `IQueryable<TEntity>` and you can freely query using the standard LINQ methods. This is fine for most of the applications. However, some ORM providers or database systems may not support standard `IQueryable` interface. If you want to use such providers, you can't rely on the `IQueryable`.

### Basic Repositories

ABP provides `IBasicRepository<TEntity, TPrimaryKey>` and `IBasicRepository<TEntity>` interfaces to support such scenarios. You can extend these interfaces (and optionally derive from `BasicRepositoryBase`) to create custom repositories for your entities.

Depending on `IBasicRepository` but not depending on `IRepository` has an advantage to make possible to work with all data sources even if they don't support `IQueryable`.

Major vendors, like Entity Framework, NHibernate or MongoDB already support `IQueryable`. So, working with `IRepository` is the **\*\*suggested\*\*** way for typical applications. But reusable module developers may consider `IBasicRepository` to support a wider range of data sources.

### Read Only Repositories

There are also `IReadOnlyRepository<TEntity, TKey>` and `IReadOnlyBasicRepository<Tentity, TKey>` interfaces for who only want to depend on querying capabilities of the repositories.

The `IReadOnlyRepository<TEntity, TKey>` derives the `IReadOnlyBasicRepository<Tentity, TKey>` and provides the following properties and methods as well:

Properties:

`AsyncExecuter`: a service that is used to execute an `IQueryable<T>` object asynchronously **\*\*without depending on the actual database provider\*\***.

Methods:

- `GetListAsync()`

- `GetQueryableAsync()`

- `WithDetails()` 1 overload

- `WithDetailsAsync()` 1 overload

Where as the `IReadOnlyBasicRepository<Tentity, TKey>` provides the following methods:

- `GetCountAsync()`

- `GetListAsync()`

- `GetPagedListAsync()`

They can all be seen as below:

![generic-repositories](images/generic-repositories.png)

#### Read Only Repositories behavior in Entity Framework Core

Entity Framework Core read-only repository implementation uses [EF Core's No-Tracking feature](https://learn.microsoft.com/en-us/ef/core/querying/tracking#no-tracking-queries). That means the entities returned from the repository will not be tracked by the EF Core [change tracker](https://learn.microsoft.com/en-us/ef/core/change-tracking/), because it is expected that you won't update entities queried from a read-only repository. If you need to track the entities, you can still uses [AsTracking()](https://learn.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.entityframeworkqueryableextensions.astracking) extension method.

> This behavior works only if the repository object is injected with one of the read-only repository interfaces (`IReadOnlyRepository<...>` or `IReadOnlyBasicRepository<...>`). It won't work if you have injected a standard repository (e.g. `IRepository<...>`) then casted it to a read-only repository interface.

### Generic Repository without a Primary Key

If your entity does not have an Id primary key (it may have a composite primary key for instance) then you cannot use the `IRepository<TEntity, TKey>` (or basic/readonly versions) defined above. In that case, you can inject and use `IRepository<TEntity>` for your entity.

> `IRepository<TEntity>` has a few missing methods those normally works with the `Id` property of an entity. Because of the entity has no `Id` property in that case, these methods are not available. One example is the `Get` method that gets an id and returns the entity with given id. However, you can still use `IQueryable<TEntity>` features to query entities by standard LINQ methods.

## Custom Repositories

Default generic repositories will be sufficient for most cases. However, you may need to create a custom repository class for your entity.

### Custom Repository Example

ABP does not force you to implement any interface or inherit from any base class for a repository. It can be just a simple POCO class. However, it's suggested to inherit existing repository interface and classes to make your work easier and get the standard methods out of the box.

#### Custom Repository Interface

First, define an interface in your domain layer:

```c#

public interface IPersonRepository : IRepository<Person, Guid>

{

Task<Person> FindByNameAsync(string name);

}

```

This interface extends `IRepository<Person, Guid>` to take advantage of pre-built repository functionality.

#### Custom Repository Implementation

A custom repository is tightly coupled to the data access tool type you are using. In this example, we will use Entity Framework Core:

````C#

public class PersonRepository : EfCoreRepository<MyDbContext, Person, Guid>, IPersonRepository

{

public PersonRepository(IDbContextProvider<TestAppDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task<Person> FindByNameAsync(string name)

{

var dbContext = await GetDbContextAsync();

return await dbContext.Set<Person>()

.Where(p => p.Name == name)

.FirstOrDefaultAsync();

}

}

````

You can directly access the data access provider (`DbContext` in this case) to perform operations.

> See [EF Core](Entity-Framework-Core.md) or [MongoDb](MongoDB.md) document for more info about the custom repositories.

## IQueryable & Async Operations

`IRepository` provides `GetQueryableAsync()` to obtain an `IQueryable`, that means you can **\*\*directly use LINQ extension methods\*\*** on it, as shown in the example of the "*\*Querying / LINQ over the Repositories\**" section above.

**\*\*Example: Using the** `Where(...)` **and the** `ToList()` **extension methods\*\***

````csharp

var queryable = await \_personRepository.GetQueryableAsync();

var people = queryable

.Where(p => p.Name.Contains(nameFilter))

.ToList();

````

`.ToList`, `Count()`... are standard extension methods defined in the `System.Linq` namespace ([see all](https://docs.microsoft.com/en-us/dotnet/api/system.linq.queryable)).

You normally want to use `.ToListAsync()`, `.CountAsync()`... instead, to be able to write a **\*\*truly async code\*\***.

However, you see that you can't use all the async extension methods in your application or domain layer when you create a new project using the standard [application startup template](Startup-Templates/Application.md), because;

\* These async methods **\*\*are not standard LINQ methods\*\*** and they are defined in the [Microsoft.EntityFrameworkCore](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore) NuGet package.

\* The standard template **\*\*doesn't have a reference\*\*** to the EF Core package from the domain and application layers, to be independent from the database provider.

Based on your requirements and development model, you have the following options to be able to use the async methods.

> Using async methods is strongly suggested! Don't use sync LINQ methods while executing database queries to be able to develop a scalable application.

### Option-1: Reference to the Database Provider Package

**\*\*The easiest solution\*\*** is to directly add the EF Core package from the project you want to use these async methods.

> Add the [Volo.Abp.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore) NuGet package to your project, which indirectly reference to the EF Core package. This ensures that you use the correct version of the EF Core compatible to the rest of your application.

When you add the NuGet package to your project, you can take full power of the EF Core extension methods.

**\*\*Example: Directly using the** `ToListAsync()` **after adding the EF Core package\*\***

````csharp

var queryable = await \_personRepository.GetQueryableAsync();

var people = queryable

.Where(p => p.Name.Contains(nameFilter))

.ToListAsync();

````

This method is suggested;

\* If you are developing an application and you **\*\*don't plan to change\*\*** EF Core in the future, or you can **\*\*tolerate\*\*** it if you need to change later. We believe that's reasonable if you are developing a final application.

#### MongoDB Case

If you are using [MongoDB](MongoDB.md), you need to add the [Volo.Abp.MongoDB](https://www.nuget.org/packages/Volo.Abp.MongoDB) NuGet package to your project. Even in this case, you can't directly use async LINQ extensions (like `ToListAsync`) because MongoDB doesn't provide async extension methods for `IQueryable<T>`, but provides for `IMongoQueryable<T>`. You need to cast the query to `IMongoQueryable<T>` first to be able to use the async extension methods.

**\*\*Example: Cast** `IQueryable<T>` **to** `IMongoQueryable<T>` **and use** `ToListAsync()`**\*\***

````csharp

var queryable = await \_personRepository.GetQueryableAsync();

var people = ((IMongoQueryable<Person>) queryable

.Where(p => p.Name.Contains(nameFilter)))

.ToListAsync();

````

### Option-2: Use the IRepository Async Extension Methods

ABP Framework provides async extension methods for the repositories, just similar to async LINQ extension methods.

**\*\*Example: Use** `CountAsync` **and** `FirstOrDefaultAsync` **methods on the repositories\*\***

````csharp

var countAll = await \_personRepository

.CountAsync();

var count = await \_personRepository

.CountAsync(x => x.Name.StartsWith("A"));

var book1984 = await \_bookRepository

.FirstOrDefaultAsync(x => x.Name == "John");

````

The standard LINQ extension methods are supported: *\*AllAsync, AnyAsync, AverageAsync, ContainsAsync, CountAsync, FirstAsync, FirstOrDefaultAsync, LastAsync, LastOrDefaultAsync, LongCountAsync, MaxAsync, MinAsync, SingleAsync, SingleOrDefaultAsync, SumAsync, ToArrayAsync, ToListAsync\**.

This approach still **\*\*has a limitation\*\***. You need to call the extension method directly on the repository object. For example, the below usage is **\*\*not supported\*\***:

```csharp

var queryable = await \_bookRepository.GetQueryableAsync();

var count = await queryable.Where(x => x.Name.Contains("A")).CountAsync();

```

This is because the `CountAsync()` method in this example is called on a `IQueryable` interface, not on the repository object. See the other options for such cases.

This method is suggested **\*\*wherever possible\*\***.

### Option-3: IAsyncQueryableExecuter

`IAsyncQueryableExecuter` is a service that is used to execute an `IQueryable<T>` object asynchronously **\*\*without depending on the actual database provider\*\***.

**\*\*Example: Inject & use the** `IAsyncQueryableExecuter.ToListAsync()` **method\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Linq;

namespace AbpDemo

{

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IRepository<Product, Guid> \_productRepository;

private readonly IAsyncQueryableExecuter \_asyncExecuter;

public ProductAppService(

IRepository<Product, Guid> productRepository,

IAsyncQueryableExecuter asyncExecuter)

{

\_productRepository = productRepository;

\_asyncExecuter = asyncExecuter;

}

public async Task<ListResultDto<ProductDto>> GetListAsync(string name)

{

//Obtain the IQueryable<T>

var queryable = await \_productRepository.GetQueryableAsync();

//Create the query

var query = queryable

.Where(p => p.Name.Contains(name))

.OrderBy(p => p.Name);

//Run the query asynchronously

List<Product> products = await \_asyncExecuter.ToListAsync(query);

//...

}

}

}

````

> `ApplicationService` and `DomainService` base classes already have `AsyncExecuter` properties pre-injected and usable without needing an explicit constructor injection.

ABP Framework executes the query asynchronously using the actual database provider's API. While that is not a usual way to execute a query, it is the best way to use the async API without depending on the database provider.

This method is suggested;

\* If you want to develop your application code **\*\*without depending\*\*** on the database provider.

\* If you are building a **\*\*reusable library\*\*** that doesn't have a database provider integration package, but needs to execute an `IQueryable<T>` object in some case.

For example, ABP Framework uses the `IAsyncQueryableExecuter` in the `CrudAppService` base class (see the [application services](Application-Services.md) document).

### Option-4: Custom Repository Methods

You can always create custom repository methods and use the database provider specific APIs, like async extension methods here. See [EF Core](Entity-Framework-Core.md) or [MongoDb](MongoDB.md) document for more info about the custom repositories.

This method is suggested;

\* If you want to **\*\*completely isolate\*\*** your domain & application layers from the database provider.

\* If you develop a **\*\*reusable [**application module**](Modules/Index.md)\*\*** and don't want to force to a specific database provider, which should be done as a [best practice](Best-Practices/Index.md).

#### Domain Services

# Domain Services

## Introduction

In a [Domain Driven Design](Domain-Driven-Design.md) (DDD) solution, the core business logic is generally implemented in aggregates ([entities](Entities.md)) and the Domain Services. Creating a Domain Service is especially needed when;

\* You implement a core domain logic that depends on some services (like repositories or other external services).

\* The logic you need to implement is related to more than one aggregate/entity, so it doesn't properly fit in any of the aggregates.

## ABP Domain Service Infrastructure

Domain Services are simple, stateless classes. While you don't have to derive from any service or interface, ABP Framework provides some useful base classes and conventions.

### DomainService & IDomainService

Either derive a Domain Service from the `DomainService` base class or directly implement the `IDomainService` interface.

**\*\*Example: Create a Domain Service deriving from the** `DomainService` **base class.\*\***

````csharp

using Volo.Abp.Domain.Services;

namespace MyProject.Issues

{

public class IssueManager : DomainService

{

}

}

````

When you do that;

\* ABP Framework automatically registers the class to the Dependency Injection system with a Transient lifetime.

\* You can directly use some common services as base properties, without needing to manually inject (e.g. [ILogger](Logging.md) and [IGuidGenerator](Guid-Generation.md)).

> It is suggested to name a Domain Service with a `Manager` or `Service` suffix. We typically use the `Manager` suffix as used in the sample above.

**\*\*Example: Implement the domain logic of assigning an Issue to a User\*\***

````csharp

public class IssueManager : DomainService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueManager(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task AssignAsync(Issue issue, AppUser user)

{

var currentIssueCount = await \_issueRepository

.CountAsync(i => i.AssignedUserId == user.Id);

//Implementing a core business validation

if (currentIssueCount >= 3)

{

throw new IssueAssignmentException(user.UserName);

}

issue.AssignedUserId = user.Id;

}

}

````

Issue is an [aggregate root](Entities.md) defined as shown below:

````csharp

public class Issue : AggregateRoot<Guid>

{

public Guid? AssignedUserId { get; internal set; }

//...

}

````

\* Making the setter `internal` ensures that it can not directly set in the upper layers and forces to always use the `IssueManager` to assign an `Issue` to a `User`.

### Using a Domain Service

A Domain Service is typically used in an [application service](Application-Services.md).

**\*\*Example: Use the** `IssueManager` **to assign an Issue to a User\*\***

````csharp

using System;

using System.Threading.Tasks;

using MyProject.Users;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace MyProject.Issues

{

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IssueManager \_issueManager;

private readonly IRepository<AppUser, Guid> \_userRepository;

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueAppService(

IssueManager issueManager,

IRepository<AppUser, Guid> userRepository,

IRepository<Issue, Guid> issueRepository)

{

\_issueManager = issueManager;

\_userRepository = userRepository;

\_issueRepository = issueRepository;

}

public async Task AssignAsync(Guid id, Guid userId)

{

var issue = await \_issueRepository.GetAsync(id);

var user = await \_userRepository.GetAsync(userId);

await \_issueManager.AssignAsync(issue, user);

await \_issueRepository.UpdateAsync(issue);

}

}

}

````

Since the `IssueAppService` is in the Application Layer, it can't directly assign an issue to a user. So, it uses the `IssueManager`.

## Application Services vs Domain Services

While both of [Application Services](Application-Services.md) and Domain Services implement the business rules, there are fundamental logical and formal differences;

\* Application Services implement the **\*\*use cases\*\*** of the application (user interactions in a typical web application), while Domain Services implement the **\*\*core, use case independent domain logic\*\***.

\* Application Services get/return [Data Transfer Objects](Data-Transfer-Objects.md), Domain Service methods typically get and return the **\*\*domain objects\*\*** ([entities](Entities.md), [value objects](Value-Objects.md)).

\* Domain services are typically used by the Application Services or other Domain Services, while Application Services are used by the Presentation Layer or Client Applications.

## Lifetime

Lifetime of Domain Services are [transient](https://docs.abp.io/en/abp/latest/Dependency-Injection) and they are automatically registered to the dependency injection system.

#### Specifications

# Specifications

Specification Pattern is used to define **\*\*named, reusable, combinable and testable filters\*\*** for entities and other business objects.

> A Specification is a part of the Domain Layer.

## Installation

> This package is **\*\*already installed\*\*** when you use the startup templates. So, most of the times you don't need to manually install it.

Install the [Volo.Abp.Specifications](https://abp.io/package-detail/Volo.Abp.Specifications) package to your project. You can use the [ABP CLI](CLI.md) *\*add-package\** command in a command line terminal when the current folder is the root folder of your project (`.csproj`):

````bash

abp add-package Volo.Abp.Specifications

````

## Defining the Specifications

Assume that you've a Customer entity as defined below:

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace MyProject

{

public class Customer : AggregateRoot<Guid>

{

public string Name { get; set; }

public byte Age { get; set; }

public long Balance { get; set; }

public string Location { get; set; }

}

}

````

You can create a new Specification class derived from the `Specification<Customer>`.

**\*\*Example: A specification to select the customers with 18+ age:\*\***

````csharp

using System;

using System.Linq.Expressions;

using Volo.Abp.Specifications;

namespace MyProject

{

public class Age18PlusCustomerSpecification : Specification<Customer>

{

public override Expression<Func<Customer, bool>> ToExpression()

{

return c => c.Age >= 18;

}

}

}

````

You simply define a lambda [Expression](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions) to define a specification.

> Instead, you can directly implement the `ISpecification<T>` interface, but the `Specification<T>` base class much simplifies it.

## Using the Specifications

There are two common use cases of the specifications.

### IsSatisfiedBy

`IsSatisfiedBy` method can be used to check if a single object satisfies the specification.

**\*\*Example: Throw exception if the customer doesn't satisfy the age specification\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class CustomerService : ITransientDependency

{

public async Task BuyAlcohol(Customer customer)

{

if (!new Age18PlusCustomerSpecification().IsSatisfiedBy(customer))

{

throw new Exception(

"This customer doesn't satisfy the Age specification!"

);

}

//TODO...

}

}

}

````

### ToExpression & Repositories

`ToExpression()` method can be used to use the specification as Expression. In this way, you can use a specification to **\*\*filter entities while querying from the database\*\***.

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

namespace MyProject

{

public class CustomerManager : DomainService, ITransientDependency

{

private readonly IRepository<Customer, Guid> \_customerRepository;

public CustomerManager(IRepository<Customer, Guid> customerRepository)

{

\_customerRepository = customerRepository;

}

public async Task<List<Customer>> GetCustomersCanBuyAlcohol()

{

var queryable = await \_customerRepository.GetQueryableAsync();

var query = queryable.Where(

new Age18PlusCustomerSpecification().ToExpression()

);

return await AsyncExecuter.ToListAsync(query);

}

}

}

````

> Specifications are correctly translated to SQL/Database queries and executed efficiently in the DBMS side. While it is not related to the Specifications, see the [Repositories](Repositories.md) document if you want to know more about the `AsyncExecuter`.

Actually, using the `ToExpression()` method is not necessary since the specifications are automatically casted to Expressions. This would also work:

````csharp

var queryable = await \_customerRepository.GetQueryableAsync();

var query = queryable.Where(

new Age18PlusCustomerSpecification()

);

````

## Composing the Specifications

One powerful feature of the specifications is that they are composable with `And`, `Or`, `Not` and `AndNot` extension methods.

Assume that you have another specification as defined below:

```csharp

using System;

using System.Linq.Expressions;

using Volo.Abp.Specifications;

namespace MyProject

{

public class PremiumCustomerSpecification : Specification<Customer>

{

public override Expression<Func<Customer, bool>> ToExpression()

{

return (customer) => (customer.Balance >= 100000);

}

}

}

```

You can combine the `PremiumCustomerSpecification` with the `Age18PlusCustomerSpecification` to query the count of premium adult customers as shown below:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

using Volo.Abp.Specifications;

namespace MyProject

{

public class CustomerManager : DomainService, ITransientDependency

{

private readonly IRepository<Customer, Guid> \_customerRepository;

public CustomerManager(IRepository<Customer, Guid> customerRepository)

{

\_customerRepository = customerRepository;

}

public async Task<int> GetAdultPremiumCustomerCountAsync()

{

return await \_customerRepository.CountAsync(

new Age18PlusCustomerSpecification()

.And(new PremiumCustomerSpecification()).ToExpression()

);

}

}

}

````

If you want to make this combination another reusable specification, you can create such a combination specification class deriving from the `AndSpecification`:

````csharp

using Volo.Abp.Specifications;

namespace MyProject

{

public class AdultPremiumCustomerSpecification : AndSpecification<Customer>

{

public AdultPremiumCustomerSpecification()

: base(new Age18PlusCustomerSpecification(),

new PremiumCustomerSpecification())

{

}

}

}

````

Now, you can re-write the `GetAdultPremiumCustomerCountAsync` method as shown below:

````csharp

public async Task<int> GetAdultPremiumCustomerCountAsync()

{

return await \_customerRepository.CountAsync(

new AdultPremiumCustomerSpecification()

);

}

````

> You see the power of the specifications with these samples. If you change the `PremiumCustomerSpecification` later, say change the balance from `100.000` to `200.000`, all the queries and combined specifications will be effected by the change. This is a good way to reduce code duplication!

## Discussions

While the specification pattern is older than C# lambda expressions, it's generally compared to expressions. Some developers may think it's not needed anymore and we can directly pass expressions to a repository or to a domain service as shown below:

````csharp

var count = await \_customerRepository.CountAsync(c => c.Balance > 100000 && c.Age => 18);

````

Since ABP's [Repository](Repositories.md) supports Expressions, this is a completely valid use. You don't have to define or use any specification in your application and you can go with expressions.

So, what's the point of a specification? Why and when should we consider to use them?

### When To Use?

Some benefits of using specifications:

- **\*\*Reusable\*\***: Imagine that you need the Premium Customer filter in many places in your code base. If you go with expressions and do not create a specification, what happens if you later change the "Premium Customer" definition? Say you want to change the minimum balance from $100,000 to $250,000 and add another condition to be a customer older than 3 years. If you'd used a specification, you just change a single class. If you repeated (copy/pasted) the same expression everywhere, you need to change all of them.

- **\*\*Composable\*\***: You can combine multiple specifications to create new specifications. This is another type of reusability.

- **\*\*Named\*\***: `PremiumCustomerSpecification` better explains the intent rather than a complex expression. So, if you have an expression that is meaningful in your business, consider using specifications.

- **\*\*Testable\*\***: A specification is a separately (and easily) testable object.

### When To Not Use?

- **\*\*Non business expressions\*\***: Do not use specifications for non business-related expressions and operations.

- **\*\*Reporting\*\***: If you are just creating a report, do not create specifications, but directly use `IQueryable` & LINQ expressions. You can even use plain SQL, views or another tool for reporting. DDD does not necessarily care about reporting, so the way you query the underlying data store can be important from a performance perspective.

### Application Layer

#### Application Services

# 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**](Data-Transfer-Objects.md))\*\*** 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:

````csharp

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, BookType type, float? price = 0)

{

Id = id;

Name = CheckName(name);

Type = type;

Price = price;

}

public virtual void ChangeName([NotNull] string name)

{

Name = CheckName(name);

}

private static string CheckName(string name)

{

if (string.IsNullOrWhiteSpace(name))

{

throw new ArgumentException(

$"name can not be empty or white space!");

}

if (name.Length > MaxNameLength)

{

throw new ArgumentException(

$"name can not be longer than {MaxNameLength} chars!");

}

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 fully 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:

````csharp

public interface IBookAppService : IApplicationService

{

Task CreateAsync(CreateBookDto input);

}

````

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

````csharp

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](Data-Transfer-Objects.md) for more about DTOs.

### BookAppService (Implementation)

````csharp

public class BookAppService : ApplicationService, IBookAppService

{

private readonly IRepository<Book, Guid> \_bookRepository;

public BookAppService(IRepository<Book, Guid> bookRepository)

{

\_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](Guid-Generation.md) document).

\* `BookAppService` implements the `IBookAppService` as expected.

\* `BookAppService` [injects](Dependency-Injection.md) `IRepository<Book, Guid>` (see [repositories](Repositories.md)) 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 get and return DTOs instead of entities. ABP does not force this rule. However, exposing entities to the presentation layer (or to remote clients) has significant problems and is not suggested.

See the [DTO documentation](Data-Transfer-Objects.md) 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](Object-To-Object-Mapping.md) infrastructure to make this even easier.

Object to object mapping provides abstractions and it is implemented by the [AutoMapper](https://automapper.org/) library by default.

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

````csharp

public interface IBookAppService : IApplicationService

{

Task CreateAsync(CreateBookDto input);

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

}

````

`BookDto` is a simple [DTO](Data-Transfer-Objects.md) class defined as below:

````csharp

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](https://docs.automapper.org/en/stable/Configuration.html#profile-instances). Example:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<Book, BookDto>();

}

}

````

You should then register profiles using the `AbpAutoMapperOptions`:

````csharp

[DependsOn(typeof(AbpAutoMapperModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAutoMapperOptions>(options =>

{

//Add all mappings defined in the assembly of the MyModule class

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](https://docs.automapper.org/en/stable/Attribute-mapping.html).

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

````csharp

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](Object-To-Object-Mapping.md) 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](Validation.md) for more.

## Authorization

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

See the [authorization document](Authorization.md) for more.

## 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.

````csharp

public interface IBookAppService :

ICrudAppService< //Defines CRUD methods

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting on getting a list of books

CreateUpdateBookDto, //Used to create a new book

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

`ICrudAppService` declares the following methods:

````csharp

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(TGetListInput input);

Task<TEntityDto> CreateAsync(TCreateInput input);

Task<TEntityDto> UpdateAsync(TKey id, TUpdateInput input);

Task DeleteAsync(TKey id);

}

````

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

````csharp

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.Undefined;

[Required]

public float Price { get; set; }

}

````

[Profile](https://docs.automapper.org/en/stable/Configuration.html#profile-instances) class of DTO class.

```csharp

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:

````csharp

public class BookAppService :

CrudAppService<Book, BookDto, Guid, PagedAndSortedResultRequestDto,

CreateUpdateBookDto, CreateUpdateBookDto>,

IBookAppService

{

public BookAppService(IRepository<Book, Guid> repository)

: 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.

### 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:

````csharp

public class DistrictAppService

: AbstractKeyCrudAppService<District, DistrictDto, DistrictKey>

{

public DistrictAppService(IRepository<District> repository)

: base(repository)

{

}

protected async override Task DeleteByIdAsync(DistrictKey id)

{

await Repository.DeleteAsync(d => d.CityId == id.CityId && d.Name == id.Name);

}

protected async override Task<District> GetEntityByIdAsync(DistrictKey id)

{

var queryable = await Repository.GetQueryableAsync();

return await AsyncQueryableExecuter.FirstOrDefaultAsync(

queryable.Where(d => d.CityId == id.CityId && d.Name == id.Name)

);

}

}

````

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

````csharp

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:

```csharp

public class MyPeopleAppService : CrudAppService<Person, PersonDto, Guid>

{

public MyPeopleAppService(IRepository<Person, Guid> repository)

: 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:

```csharp

public class MyPeopleAppService : CrudAppService<Person, PersonDto, Guid>

{

public MyPeopleAppService(IRepository<Person, Guid> repository)

: base(repository)

{

}

protected async override Task CheckDeletePolicyAsync()

{

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:

````csharp

Task<TGetOutputDto> GetAsync(TKey id);

Task<PagedResultDto<TGetListOutputDto>> GetListAsync(TGetListInput input);

Task<TGetOutputDto> CreateAsync(TCreateInput input);

Task<TGetOutputDto> UpdateAsync(TKey id, TUpdateInput input);

Task DeleteAsync(TKey id);

````

#### Querying

These methods are low level methods that can 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 creation 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 use the [IObjectMapper](Object-To-Object-Mapping.md) 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\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Content;

namespace MyProject.Test

{

public interface ITestAppService : IApplicationService

{

Task Upload(Guid id, IRemoteStreamContent streamContent);

Task<IRemoteStreamContent> Download(Guid id);

Task CreateFile(CreateFileInput input);

Task CreateMultipleFile(CreateMultipleFileInput input);

}

public class CreateFileInput

{

public Guid Id { get; set; }

public IRemoteStreamContent Content { get; set; }

}

public class CreateMultipleFileInput

{

public Guid Id { get; set; }

public IEnumerable<IRemoteStreamContent> Contents { get; set; }

}

}

````

**\*\*You need to configure** `AbpAspNetCoreMvcOptions` **to add DTO class to** `FormBodyBindingIgnoredTypes` **to use** `IRemoteStreamContent` **in\*\*** **\*\*DTO ([**Data Transfer Object**](Data-Transfer-Objects.md))\*\***

````csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers.FormBodyBindingIgnoredTypes.Add(typeof(CreateFileInput));

options.ConventionalControllers.FormBodyBindingIgnoredTypes.Add(typeof(CreateMultipleFileInput));

});

````

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

````csharp

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, ITestAppService

{

public Task<IRemoteStreamContent> Download(Guid id)

{

var fs = new FileStream("C:\\Temp\\" + id + ".blob", FileMode.OpenOrCreate);

return Task.FromResult(

(IRemoteStreamContent) new RemoteStreamContent(fs) {

ContentType = "application/octet-stream"

}

);

}

public async Task Upload(Guid id, IRemoteStreamContent streamContent)

{

using (var fs = new FileStream("C:\\Temp\\" + id + ".blob", FileMode.Create))

{

await streamContent.GetStream().CopyToAsync(fs);

await fs.FlushAsync();

}

}

public async Task CreateFileAsync(CreateFileInput input)

{

using (var fs = new FileStream("C:\\Temp\\" + input.Id + ".blob", FileMode.Create))

{

await input.Content.GetStream().CopyToAsync(fs);

await fs.FlushAsync();

}

}

public async Task CreateMultipleFileAsync(CreateMultipleFileInput input)

{

using (var fs = new FileStream("C:\\Temp\\" + input.Id + ".blob", FileMode.Append))

{

foreach (var content in input.Contents)

{

await content.GetStream().CopyToAsync(fs);

}

await fs.FlushAsync();

}

}

}

}

````

`IRemoteStreamContent` is compatible with the [Auto API Controller](API/Auto-API-Controllers.md) and [Dynamic C# HTTP Proxy](API/Dynamic-CSharp-API-Clients.md) systems.

## Lifetime

Lifetime of application services are [transient](Dependency-Injection.md) and they are automatically registered to the dependency injection system.

#### Data Transfer Objects

# Data Transfer Objects

## Introduction

**\*\*Data Transfer Objects\*\*** (DTO) are used to transfer data between the **\*\*Application Layer\*\*** and the **\*\*Presentation Layer\*\*** or other type of clients.

Typically, an [application service](Application-Services.md) is called from the presentation layer (optionally) with a **\*\*DTO\*\*** 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.

### The Need for DTOs

> **\*\*You can skip this section\*\*** if you feel that you know and confirm the benefits of using DTOs.

At first, creating a DTO class for each application service method can be seen as tedious and time-consuming work. However, they can save your application if you correctly use them. Why & how?

#### Abstraction of the Domain Layer

DTOs provide an efficient way of **\*\*abstracting domain objects\*\*** from the presentation layer. In effect, your **\*\*layers\*\*** are correctly separated. If you want to change the presentation layer completely, you can continue with the existing application and domain layers. Alternatively, you can re-write your domain layer, completely change the database schema, entities and O/RM framework, all without changing the presentation layer. This, of course, is as long as the contracts (method signatures and DTOs) of your application services remain unchanged.

#### Data Hiding

Say you have a `User` entity with the properties Id, Name, EmailAddress and Password. If a `GetAllUsers()` method of a `UserAppService` returns a `List<User>`, anyone can access the passwords of all your users, even if you do not show it on the screen. It's not just about security, it's about data hiding. Application services should return only what it needs by the presentation layer (or client). Not more, not less.

#### Serialization & Lazy Load Problems

When you return data (an object) to the presentation layer, it's most likely serialized. For example, in a REST API that returns JSON, your object will be serialized to JSON and sent to the client. Returning an Entity to the presentation layer can be problematic in that regard, especially if you are using a relational database and an ORM provider like Entity Framework Core. How?

In a real-world application, your entities may have references to each other. The `User` entity can have a reference to it's `Role`s. If you want to serialize `User`, its `Role`s are also serialized. The `Role` class may have a `List<Permission>` and the `Permission` class can has a reference to a `PermissionGroup` class and so on... Imagine all of these objects being serialized at once. You could easily and accidentally serialize your whole database! Also, if your objects have circular references, they may **\*\*not\*\*** be serialized at all.

What's the solution? Marking properties as `NonSerialized`? No, you can not know when it should be serialized and when it shouldn't be. It may be needed in one application service method, and not needed in another. Returning safe, serializable, and specially designed DTOs is a good choice in this situation.

Almost all O/RM frameworks support lazy-loading. It's a feature that loads entities from the database when they're needed. Say a `User` class has a reference to a `Role` class. When you get a `User` from the database, the `Role` property (or collection) is not filled. When you first read the `Role` property, it's loaded from the database. So, if you return such an Entity to the presentation layer, it will cause it to retrieve additional entities from the database by executing additional queries. If a serialization tool reads the entity, it reads all properties recursively and again your whole database can be retrieved (if there are relations between entities).

More problems can arise if you use Entities in the presentation layer. **\*\*It's best not to reference the domain/business layer assembly in the presentation layer.\*\***

If you are convinced about using DTOs, we can continue to what ABP Framework provides and suggests about DTOs.

> ABP doesn't force you to use DTOs, however using DTOs is **\*\*strongly suggested as a best practice\*\***.

## Standard Interfaces & Base Classes

A DTO is a simple class that has no dependency and you can design it in any way. However, ABP introduces some **\*\*interfaces\*\*** to determine the **\*\*conventions\*\*** for naming **\*\*standard properties\*\*** and **\*\*base classes\*\*** to **\*\*don't repeat yourself\*\*** while declaring **\*\*common properties\*\***.

**\*\*None of them are required\*\***, but using them **\*\*simplifies and standardizes\*\*** your application code.

### Entity Related DTOs

You typically create DTOs corresponding to your entities, which results similar classes to your entities. ABP Framework provides some base classes to simplify while creating such DTOs.

#### EntityDto

`IEntityDto<TKey>` is a simple interface that only defines an `Id` property. You can implement it or inherit from the `EntityDto<TKey>` for your DTOs that matches to an [entity](Entities.md).

**\*\*Example:\*\***

````csharp

using System;

using Volo.Abp.Application.Dtos;

namespace AbpDemo

{

public class ProductDto : EntityDto<Guid>

{

public string Name { get; set; }

//...

}

}

````

#### Audited DTOs

If your entity inherits from audited entity classes (or implements auditing interfaces), you can use the following base classes to create your DTOs:

\* `CreationAuditedEntityDto`

\* `CreationAuditedEntityWithUserDto`

\* `AuditedEntityDto`

\* `AuditedEntityWithUserDto`

\* `FullAuditedEntityDto`

\* `FullAuditedEntityWithUserDto`

#### Extensible DTOs

If you want to use the [object extension system](Object-Extensions.md) for your DTOs, you can use or inherit from the following DTO classes:

\* `ExtensibleObject` implements the `IHasExtraProperties` (other classes inherits this class).

\* `ExtensibleEntityDto`

\* `ExtensibleCreationAuditedEntityDto`

\* `ExtensibleCreationAuditedEntityWithUserDto`

\* `ExtensibleAuditedEntityDto`

\* `ExtensibleAuditedEntityWithUserDto`

\* `ExtensibleFullAuditedEntityDto`

\* `ExtensibleFullAuditedEntityWithUserDto`

### List Results

It is common to return a list of DTOs to the client. `IListResult<T>` interface and `ListResultDto<T>` class is used to make it standard.

The definition of the `IListResult<T>` interface:

````csharp

public interface IListResult<T>

{

IReadOnlyList<T> Items { get; set; }

}

````

**\*\*Example: Return a list of products\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace AbpDemo

{

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IRepository<Product, Guid> \_productRepository;

public ProductAppService(IRepository<Product, Guid> productRepository)

{

\_productRepository = productRepository;

}

public async Task<ListResultDto<ProductDto>> GetListAsync()

{

//Get entities from the repository

List<Product> products = await \_productRepository.GetListAsync();

//Map entities to DTOs

List<ProductDto> productDtos =

ObjectMapper.Map<List<Product>, List<ProductDto>>(products);

//Return the result

return new ListResultDto<ProductDto>(productDtos);

}

}

}

````

You could simply return the `productDtos` object (and change the method return type) and it has nothing wrong. Returning a `ListResultDto` makes your `List<ProductDto>` wrapped into another object as an `Items` property. This has one advantage: You can later add more properties to your return value without breaking your remote clients (when they get the value as a JSON result). So, it is especially suggested when you are developing reusable application modules.

### Paged & Sorted List Results

It is more common to request a paged list from server and return a paged list to the client. ABP defines a few interface and classes to standardize it:

#### Input (Request) Types

The following interfaces and classes is to standardize the input sent by the clients.

\* `ILimitedResultRequest`: Defines a `MaxResultCount` (`int`) property to request a limited result from the server.

\* `IPagedResultRequest`: Inherits from the `ILimitedResultRequest` (so it inherently has the `MaxResultCount` property) and defines a `SkipCount` (`int`) to declare the skip count while requesting a paged result from the server.

\* `ISortedResultRequest`: Defines a `Sorting` (`string`) property to request a sorted result from the server. Sorting value can be "*\*Name\**", "*\*Name DESC\**", "*\*Name ASC, Age DESC\**"... etc.

\* `IPagedAndSortedResultRequest` inherits from both of the `IPagedResultRequest` and `ISortedResultRequest`, so has `MaxResultCount`, `SkipCount` and `Sorting` properties.

Instead of implementing the interfaces manually, it is suggested to inherit one of the following base DTO classes:

\* `LimitedResultRequestDto` implements `ILimitedResultRequest`.

\* `PagedResultRequestDto` implements `IPagedResultRequest` (and inherits from the `LimitedResultRequestDto`).

\* `PagedAndSortedResultRequestDto` implements `IPagedAndSortedResultRequest` (and inherit from the `PagedResultRequestDto`).

##### Max Result Count

`LimitedResultRequestDto` (and inherently the others) limits and validates the `MaxResultCount` by the following rules;

\* If the client doesn't set `MaxResultCount`, it is assumed as **\*\*10\*\*** (the default page size). This value can be changed by setting the `LimitedResultRequestDto.DefaultMaxResultCount` static property.

\* If the client sends `MaxResultCount` greater than **\*\*1,000\*\***, it produces a **\*\*validation error\*\***. It is important to protect the server from abuse of the service. If you want, you can change this value by setting the `LimitedResultRequestDto.MaxMaxResultCount` static property.

Static properties suggested to be set on application startup since they are static (global).

#### Output (Response) Types

The following interfaces and classes is to standardize the output sent to the clients.

\* `IHasTotalCount` defines a `TotalCount` (`long`) property to return the total count of the records in case of paging.

\* `IPagedResult<T>` inherits from the `IListResult<T>` and `IHasTotalCount`, so it has the `Items` and `TotalCount` properties.

Instead of implementing the interfaces manually, it is suggested to inherit one of the following base DTO classes:

\* `PagedResultDto<T>` inherits from the `ListResultDto<T>` and also implements the `IPagedResult<T>`.

**\*\*Example: Request a paged & sorted result from server and return a paged list\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace AbpDemo

{

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IRepository<Product, Guid> \_productRepository;

public ProductAppService(IRepository<Product, Guid> productRepository)

{

\_productRepository = productRepository;

}

public async Task<PagedResultDto<ProductDto>> GetListAsync(

PagedAndSortedResultRequestDto input)

{

//Create the query

var query = \_productRepository

.OrderBy(input.Sorting);

//Get total count from the repository

var totalCount = await query.CountAsync();

//Get entities from the repository

List<Product> products = await query

.Skip(input.SkipCount)

.Take(input.MaxResultCount).ToListAsync();

//Map entities to DTOs

List<ProductDto> productDtos =

ObjectMapper.Map<List<Product>, List<ProductDto>>(products);

//Return the result

return new PagedResultDto<ProductDto>(totalCount, productDtos);

}

}

}

````

ABP Framework also defines a `PageBy` extension method (that is compatible with the `IPagedResultRequest`) that can be used instead of `Skip` + `Take` calls:

````csharp

var query = \_productRepository

.OrderBy(input.Sorting)

.PageBy(input);

````

> Notice that we added `Volo.Abp.EntityFrameworkCore` package to the project to be able to use the `ToListAsync` and `CountAsync` methods since they are not included in the standard LINQ, but defined by the Entity Framework Core.

See also the [repository documentation](Repositories.md) to if you haven't understood the example code.

## Related Topics

### Validation

Inputs of [application service](Application-Services.md) methods, controller actions, page model inputs... are automatically validated. You can use the standard data annotation attributes or a custom validation method to perform the validation.

See the [validation document](Validation.md) for more.

### Object to Object Mapping

When you create a DTO that is related to an entity, you generally need to map these objects. ABP provides an object to object mapping system to simplify the mapping process. See the following documents:

\* [Object to Object Mapping document](Object-To-Object-Mapping.md) covers all the features.

\* [Application Services document](Application-Services.md) provides a full example.

## Best Practices

You are free to design your DTO classes. However, there are some best practices & suggestions that you may want to follow.

### Common Principles

\* DTOs should be **\*\*well serializable\*\*** since they are generally serialized and deserialized (to JSON or other format). It is suggested to have an empty (parameterless) public constructor if you have another constructor with parameter(s).

\* DTOs **\*\*should not contain any business logic\*\***, except some formal [validation](Validation.md) code.

\* Do not inherit DTOs from entities and **\*\*do not reference to entities\*\***. The [application startup template](Startup-Templates/Application.md) already prevents it by separating the projects.

\* If you use an auto [object to object mapping](Object-To-Object-Mapping.md) library, like AutoMapper, enable the **\*\*mapping configuration validation\*\*** to prevent potential bugs.

### Input DTO Principles

\* Define only the **\*\*properties needed\*\*** for the use case. Do not include properties not used for the use case, which confuses developers if you do so.

\* **\*\*Don't reuse\*\*** input DTOs among different application service methods. Because, different use cases will need to and use different properties of the DTO which results some properties are not used in some cases and that makes harder to understand and use the services and causes potential bugs in the future.

### Output DTO Principles

\* You can **\*\*reuse output DTOs\*\*** if you **\*\*fill all the properties\*\*** on all the cases.

#### Unit Of Work

# Unit of Work

ABP Framework's Unit Of Work (UOW) implementation provides an abstraction and control on a **\*\*database connection and transaction\*\*** scope in an application.

Once a new UOW started, it creates an **\*\*ambient scope\*\*** that is participated by **\*\*all the database operations\*\*** performed in the current scope and considered as a **\*\*single transaction boundary\*\***. The operations are **\*\*committed\*\*** (on success) or **\*\*rolled back\*\*** (on exception) all together.

ABP's UOW system is;

\* **\*\*Works conventional\*\***, so most of the times you don't deal with UOW at all.

\* **\*\*Database provider independent\*\***.

\* **\*\*Web independent\*\***, that means you can create unit of work scopes in any type of applications beside web applications/services.

## Conventions

The following method types are considered as a unit of work:

\* ASP.NET Core MVC **\*\*Controller Actions\*\***.

\* ASP.NET Core Razor **\*\*Page Handlers\*\***.

\* **\*\*Application service\*\*** methods.

\* **\*\*Repository methods\*\***.

A UOW automatically begins for these methods **\*\*except\*\*** if there is already a **\*\*surrounding (ambient)\*\*** UOW in action. Examples;

\* If you call a [repository](Repositories.md) method and there is no UOW started yet, it automatically **\*\*begins a new transactional UOW\*\*** that involves all the operations done in the repository method and **\*\*commits the transaction\*\*** if the repository method **\*\*doesn't throw any exception.\*\*** The repository method doesn't know about UOW or transaction at all. It just works on a regular database objects (`DbContext` for [EF Core](Entity-Framework-Core.md), for example) and the UOW is handled by the ABP Framework.

\* If you call an [application service](Application-Services.md) method, the same UOW system works just as explained above. If the application service method uses some repositories, the repositories **\*\*don't begin a new UOW\*\***, but **\*\*participates to the current unit of work\*\*** started by the ABP Framework for the application service method.

\* The same is true for an ASP.NET Core controller action. If the operation has started with a controller action, then the **\*\*UOW scope is the controller action's method body\*\***.

All of these are automatically handled by the ABP Framework.

### Database Transaction Behavior

While the section above explains the UOW as it is database transaction, actually a UOW doesn't have to be transactional. By default;

\* **\*\*HTTP GET\*\*** requests don't start a transactional UOW. They still starts a UOW, but **\*\*doesn't create a database transaction\*\***.

\* All other HTTP request types start a UOW with a database transaction, if database level transactions are supported by the underlying database provider.

This is because an HTTP GET request doesn't (and shouldn't) make any change in the database. You can change this behavior using the options explained below.

## Default Options

`AbpUnitOfWorkDefaultOptions` is used to configure the default options for the unit of work system. Configure the options in the `ConfigureServices` method of your [module](Module-Development-Basics.md).

**\*\*Example: Completely disable the database transactions\*\***

````csharp

Configure<AbpUnitOfWorkDefaultOptions>(options =>

{

options.TransactionBehavior = UnitOfWorkTransactionBehavior.Disabled;

});

````

### Option Properties

\* `TransactionBehavior` (`enum`: `UnitOfWorkTransactionBehavior`). A global point to configure the transaction behavior. Default value is `Auto` and work as explained in the "*\*Database Transaction Behavior\**" section above. You can enable (even for HTTP GET requests) or disable transactions with this option.

\* `TimeOut` (`int?`): Used to set the timeout value for UOWs. **\*\*Default value is** `null`**\*\*** and uses to the default of the underlying database provider.

\* `IsolationLevel` (`IsolationLevel?`): Used to set the [isolation level](https://docs.microsoft.com/en-us/dotnet/api/system.data.isolationlevel) of the database transaction, if the UOW is transactional.

## Controlling the Unit Of Work

In some cases, you may want to change the conventional transaction scope, create inner scopes or fine control the transaction behavior. The following sections cover these possibilities.

### IUnitOfWorkEnabled Interface

This is an easy way to enable UOW for a class (or a hierarchy of classes) that is not unit of work by the conventions explained above.

**\*\*Example: Implement** `IUnitOfWorkEnabled` **for an arbitrary service\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyService : ITransientDependency, IUnitOfWorkEnabled

{

public virtual async Task FooAsync()

{

//this is a method with a UOW scope

}

}

}

````

Then `MyService` (and any class derived from it) methods will be UOW.

However, there are **\*\*some rules should be followed\*\*** in order to make it working;

\* If you are **\*\*not injecting\*\*** the service over an interface (like `IMyService`), then the methods of the service must be `virtual` (otherwise, [dynamic proxy / interception](Dynamic-Proxying-Interceptors.md) system can not work).

\* Only `async` methods (methods returning a `Task` or `Task<T>`) are intercepted. So, sync methods can not start a UOW.

> Notice that if `FooAsync` is called inside a UOW scope, then it already participates to the UOW without needing to the `IUnitOfWorkEnabled` or any other configuration.

### UnitOfWorkAttribute

`UnitOfWork` attribute provides much more possibility like enabling or disabling UOW and controlling the transaction behavior.

`UnitOfWork` attribute can be used for a **\*\*class\*\*** or a **\*\*method\*\*** level.

**\*\*Example: Enable UOW for a specific method of a class\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

[UnitOfWork]

public virtual async Task FooAsync()

{

//this is a method with a UOW scope

}

public virtual async Task BarAsync()

{

//this is a method without UOW

}

}

}

````

**\*\*Example: Enable UOW for all the methods of a class\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

[UnitOfWork]

public class MyService : ITransientDependency

{

public virtual async Task FooAsync()

{

//this is a method with a UOW scope

}

public virtual async Task BarAsync()

{

//this is a method with a UOW scope

}

}

}

````

Again, the **\*\*same rules\*\*** are valid here:

\* If you are **\*\*not injecting\*\*** the service over an interface (like `IMyService`), then the methods of the service must be `virtual` (otherwise, [dynamic proxy / interception](Dynamic-Proxying-Interceptors.md) system can not work).

\* Only `async` methods (methods returning a `Task` or `Task<T>`) are intercepted. So, sync methods can not start a UOW.

#### UnitOfWorkAttribute Properties

\* `IsTransactional` (`bool?`): Used to set whether the UOW should be transactional or not. **\*\*Default value is** `null`**\*\***. if you leave it `null`, it is determined automatically based on the conventions and the configuration.

\* `TimeOut` (`int?`): Used to set the timeout value for this UOW. **\*\*Default value is** `null`**\*\*** and fallbacks to the default configured value.

\* `IsolationLevel` (`IsolationLevel?`): Used to set the [isolation level](https://docs.microsoft.com/en-us/dotnet/api/system.data.isolationlevel) of the database transaction, if the UOW is transactional. If not set, uses the default configured value.

\* `IsDisabled` (`bool`): Used to disable the UOW for the current method/class.

> If a method is called in an ambient UOW scope, then the `UnitOfWork` attribute is ignored and the method participates to the surrounding transaction in any way.

**\*\*Example: Disable UOW for a controller action\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.Uow;

namespace AbpDemo.Web

{

public class MyController : AbpController

{

[UnitOfWork(IsDisabled = true)]

public virtual async Task FooAsync()

{

//...

}

}

}

````

## IUnitOfWorkManager

`IUnitOfWorkManager` is the main service that is used to control the unit of work system. The following sections explains how to directly work with this service (while most of the times you won't need).

### Begin a New Unit Of Work

`IUnitOfWorkManager.Begin` method is used to create a new UOW scope.

**\*\*Example: Create a new non-transactional UOW scope\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public MyService(IUnitOfWorkManager unitOfWorkManager)

{

\_unitOfWorkManager = unitOfWorkManager;

}

public virtual async Task FooAsync()

{

using (var uow = \_unitOfWorkManager.Begin(

requiresNew: true, isTransactional: false

))

{

//...

await uow.CompleteAsync();

}

}

}

}

````

`Begin` method gets the following optional parameters:

\* `requiresNew` (`bool`): Set `true` to ignore the surrounding unit of work and start a new UOW with the provided options. **\*\*Default value is** `false`**. If it is** `false` **and there is a surrounding UOW,** `Begin` **method doesn't actually begin a new UOW, but silently participates to the existing UOW.\*\***

\* `isTransactional` (`bool`). Default value is `false`.

\* `isolationLevel` (`IsolationLevel?`): Used to set the [isolation level](https://docs.microsoft.com/en-us/dotnet/api/system.data.isolationlevel) of the database transaction, if the UOW is transactional. If not set, uses the default configured value.

\* `TimeOut` (`int?`): Used to set the timeout value for this UOW. **\*\*Default value is** `null`**\*\*** and fallbacks to the default configured value.

### The Current Unit Of Work

UOW is ambient, as explained before. If you need to access to the current unit of work, you can use the `IUnitOfWorkManager.Current` property.

**\*\*Example: Get the current UOW\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyProductService : ITransientDependency

{

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public MyProductService(IUnitOfWorkManager unitOfWorkManager)

{

\_unitOfWorkManager = unitOfWorkManager;

}

public async Task FooAsync()

{

var uow = \_unitOfWorkManager.Current;

//...

}

}

}

````

`Current` property returns a `IUnitOfWork` object.

> **\*\*Current Unit Of Work can be** `null`**\*\*** if there is no surrounding unit of work. It won't be `null` if your class is a conventional UOW class, you manually made it UOW or it was called inside a UOW scope, as explained before.

#### SaveChangesAsync

`IUnitOfWork.SaveChangesAsync()` method can be needed to save all the changes until now to the database. If you are using EF Core, it behaves exactly same. If the current UOW is transactional, even saved changes can be rolled back on an error (for the supporting database providers).

**\*\*Example: Save changes after inserting an entity to get its auto-increment id\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace AbpDemo

{

public class CategoryAppService : ApplicationService, ICategoryAppService

{

private readonly IRepository<Category, int> \_categoryRepository;

public CategoryAppService(IRepository<Category, int> categoryRepository)

{

\_categoryRepository = categoryRepository;

}

public async Task<int> CreateAsync(string name)

{

var category = new Category {Name = name};

await \_categoryRepository.InsertAsync(category);

//Saving changes to be able to get the auto increment id

await UnitOfWorkManager.Current.SaveChangesAsync();

return category.Id;

}

}

}

````

This example uses auto-increment `int` primary key for the `Category` [entity](Entities.md). Auto-increment PKs require to save the entity to the database to get the id of the new entity.

This example is an [application service](Application-Services.md) derived from the base `ApplicationService` class, which already has the `IUnitOfWorkManager` service injected as the `UnitOfWorkManager` property. So, no need to inject it manually.

Since getting the current UOW is pretty common, there is also a `CurrentUnitOfWork` property as a shortcut to the `UnitOfWorkManager.Current`. So, the example above can be changed to use it:

````csharp

await CurrentUnitOfWork.SaveChangesAsync();

````

##### Alternative to the SaveChanges()

Since saving changes after inserting, updating or deleting an entity can be frequently needed, corresponding [repository](Repositories.md) methods has an optional `autoSave` parameter. So, the `CreateAsync` method above could be re-written as shown below:

````csharp

public async Task<int> CreateAsync(string name)

{

var category = new Category {Name = name};

await \_categoryRepository.InsertAsync(category, autoSave: true);

return category.Id;

}

````

If your intent is just to save the changes after creating/updating/deleting an entity, it is suggested to use the `autoSave` option instead of manually using the `CurrentUnitOfWork.SaveChangesAsync()`.

> **\*\*Note-1\*\***: All changes are automatically saved when a unit of work ends without any error. So, don't call `SaveChangesAsync()` and don't set `autoSave` to `true` unless you really need it.

>

> **\*\*Note-2\*\***: If you use `Guid` as the primary key, you never need to save changes on insert to just get the generated id, because `Guid` keys are set in the application and are immediately available once you create a new entity.

#### Other IUnitOfWork Properties/Methods

\* `OnCompleted` method gets a callback action which is called when the unit of work successfully completed (where you can be sure that all changes are saved).

\* `Failed` and `Disposed` events can be used to be notified if the UOW fails or when it is disposed.

\* `Complete` and `Rollback` methods are used to complete (commit) or roll backs the current UOW, which are normally used internally by the ABP Framework but can be used if you manually start a transaction using the `IUnitOfWorkManager.Begin` method.

\* `Options` can be used to get options that was used while starting the UOW.

\* `Items` dictionary can be used to store and get arbitrary objects inside the same unit of work, which can be a point to implement custom logics.

## ASP.NET Core Integration

Unit of work system is fully integrated to the ASP.NET Core. It properly works when you use ASP.NET Core MVC Controllers or Razor Pages. It defines action filters and page filters for the UOW system.

> You typically do nothing to configure the UOW when you use ASP.NET Core.

### Unit Of Work Middleware

`AbpUnitOfWorkMiddleware` is a middleware that can enable UOW in the ASP.NET Core request pipeline. This might be needed if you need to enlarge the UOW scope to cover some other middleware(s).

**\*\*Example:\*\***

````csharp

app.UseUnitOfWork();

app.UseConfiguredEndpoints();

````

### E-Book: Implementing DDD

"https://abp.io/books/implementing-domain-driven-design"

## Multi Tenancy

# Multi-Tenancy

Multi-Tenancy is a widely used architecture to create **\*\*SaaS applications\*\*** where the hardware and software **\*\*resources are shared by the customers\*\*** (tenants). ABP Framework provides all the base functionalities to create **\*\*multi tenant applications\*\***.

Wikipedia [defines](https://en.wikipedia.org/wiki/Multitenancy) the multi-tenancy as like that:

> Software **\*\*Multi-tenancy\*\*** refers to a software **\*\*architecture\*\*** in which a **\*\*single instance\*\*** of software runs on a server and serves **\*\*multiple tenants\*\***. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a software application is designed to provide every tenant a **\*\*dedicated share of the instance including its data\*\***, configuration, user management, tenant individual functionality and non-functional properties. Multi-tenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants.

## Terminology: Host vs Tenant

There are two main side of a typical SaaS / Multi-tenant application:

\* A **\*\*Tenant\*\*** is a customer of the SaaS application that pays money to use the service.

\* **\*\*Host\*\*** is the company that owns the SaaS application and manages the system.

The Host and the Tenant terms will be used for that purpose in the rest of the document.

## Configuration

### AbpMultiTenancyOptions: Enable/Disable Multi-Tenancy

`AbpMultiTenancyOptions` is the main options class to **\*\*enable/disable the multi-tenancy\*\*** for your application.

**\*\*Example: Enable multi-tenancy\*\***

```csharp

Configure<AbpMultiTenancyOptions>(options =>

{

options.IsEnabled = true;

});

```

> Multi-Tenancy is disabled in the ABP Framework by default. However, it is **\*\*enabled by default\*\*** when you create a new solution using the [startup template](Startup-Templates/Application.md). `MultiTenancyConsts` class in the solution has a constant to control it in a single place.

### Database Architecture

ABP Framework supports all the following approaches to store the tenant data in the database;

\* **\*\*Single Database\*\***: All tenants are stored in a single database.

\* **\*\*Database per Tenant\*\***: Every tenant has a separate, dedicated database to store the data related to that tenant.

\* **\*\*Hybrid\*\***: Some tenants share a single databases while some tenants may have their own databases.

[Tenant management module](Modules/Tenant-Management.md) (which comes pre-installed with the startup projects) allows you to set a connection string for any tenant (as optional), so you can achieve any of the approaches.

## Usage

Multi-tenancy system is designed to **\*\*work seamlessly\*\*** and make your application code **\*\*multi-tenancy unaware\*\*** as much as possible.

### IMultiTenant

You should implement the `IMultiTenant` interface for your [entities](Entities.md) to make them **\*\*multi-tenancy ready\*\***.

**\*\*Example: A multi-tenant *\*Product\** entity\*\***

````csharp

using System;

using Volo.Abp.Domain.Entities;

using Volo.Abp.MultiTenancy;

namespace MultiTenancyDemo.Products

{

public class Product : AggregateRoot<Guid>, IMultiTenant

{

public Guid? TenantId { get; set; } //Defined by the IMultiTenant interface

public string Name { get; set; }

public float Price { get; set; }

}

}

````

`IMultiTenant` interface just defines a `TenantId` property. When you implement this interface, ABP Framework **\*\*automatically\*\*** [filters](Data-Filtering.md) entities for the current tenant when you query from database. So, you don't need to manually add `TenantId` condition while performing queries. A tenant can not access to data of another tenant by default.

#### Why the TenantId Property is Nullable?

`IMultiTenant.TenantId` is **\*\*nullable\*\***. When it is null that means the entity is owned by the **\*\*Host\*\*** side and not owned by a tenant. It is useful when you create a functionality in your system that is both used by the tenant and the host sides.

For example, `IdentityUser` is an entity defined by the [Identity Module](Modules/Identity.md). The host and all the tenants have their own users. So, for the host side, users will have a `null` `TenantId` while tenant users will have their related `TenantId`.

> **\*\*Tip\*\***: If your entity is tenant-specific and has no meaning in the host side, you can force to not set `null` for the `TenantId` in the constructor of your entity.

#### When to set the TenantId?

ABP automatically sets the `TenantId` for you when you create a new entity object. It is done in the constructor of the base `Entity` class (all other base entity and aggregate root classes are derived from the `Entity` class). The `TenantId` is set from the current value of the `ICurrentTenant.Id` property (see the next section).

If you set the `TenantId` value for a specific entity object, it will override the value set by the base class. If you want to set the `TenantId` property yourself, we recommend to do it in the constructor of your entity class and do not change (update) it again (Actually, changing it means that you are moving the entity from a tenant to another tenant. If you want that, you need an extra care about the related entities in the database).

### ICurrentTenant

`ICurrentTenant` is the main service to interact with the multi-tenancy infrastructure.

`ApplicationService`, `DomainService`, `AbpController` and some other base classes already has pre-injected `CurrentTenant` properties. For other type of classes, you can inject the `ICurrentTenant` into your service.

#### Tenant Properties

`ICurrentTenant` defines the following properties;

\* `Id` (`Guid`): Id of the current tenant. Can be `null` if the current user is a host user or the tenant could not be determined from the request.

\* `Name` (`string`): Name of the current tenant. Can be `null` if the current user is a host user or the tenant could not be determined from the request.

\* `IsAvailable` (`bool`): Returns `true` if the `Id` is not `null`.

#### Change the Current Tenant

ABP Framework automatically filters the resources (database, cache...) based on the `ICurrentTenant.Id`. However, in some cases you may want to perform an operation on behalf of a specific tenant, generally when you are in the host context.

`ICurrentTenant.Change` method changes the current tenant for a limited scope, so you can safely perform operations for the tenant.

**\*\*Example: Get product count of a specific tenant\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

namespace MultiTenancyDemo.Products

{

public class ProductManager : DomainService

{

private readonly IRepository<Product, Guid> \_productRepository;

public ProductManager(IRepository<Product, Guid> productRepository)

{

\_productRepository = productRepository;

}

public async Task<long> GetProductCountAsync(Guid? tenantId)

{

using (CurrentTenant.Change(tenantId))

{

return await \_productRepository.GetCountAsync();

}

}

}

}

````

\* `Change` method can be used in a **\*\*nested way\*\***. It restores the `CurrentTenant.Id` to the previous value after the `using` statement.

\* When you use `CurrentTenant.Id` inside the `Change` scope, you get the `tenantId` provided to the `Change` method. So, the repository also get this `tenantId` and can filter the database query accordingly.

\* Use `CurrentTenant.Change(null)` to change scope to the host context.

> Always use the `Change` method with a `using` statement like done in this example.

### Data Filtering: Disable the Multi-Tenancy Filter

As mentioned before, ABP Framework handles data isolation between tenants using the [Data Filtering](Data-Filtering.md) system. In some cases, you may want to disable it and perform a query on all the data, without filtering for the current tenant.

**\*\*Example: Get count of products in the database, including all the products of all the tenants.\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Data;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

using Volo.Abp.MultiTenancy;

namespace MultiTenancyDemo.Products

{

public class ProductManager : DomainService

{

private readonly IRepository<Product, Guid> \_productRepository;

private readonly IDataFilter \_dataFilter;

public ProductManager(

IRepository<Product, Guid> productRepository,

IDataFilter dataFilter)

{

\_productRepository = productRepository;

\_dataFilter = dataFilter;

}

public async Task<long> GetProductCountAsync()

{

using (\_dataFilter.Disable<IMultiTenant>())

{

return await \_productRepository.GetCountAsync();

}

}

}

}

````

See the [Data Filtering document](Data-Filtering.md) for more.

> Note that this approach won't work if your tenants have **\*\*separate databases\*\*** since there is no built-in way to query from multiple database in a single database query. You should handle it yourself if you need it.

## Infrastructure

### Determining the Current Tenant

The first thing for a multi-tenant application is to determine the current tenant on the runtime.

ABP Framework provides an extensible **\*\*Tenant Resolving\*\*** system for that purpose. Tenant Resolving system then used in the **\*\*Multi-Tenancy Middleware\*\*** to determine the current tenant for the current HTTP request.

#### Tenant Resolvers

##### Default Tenant Resolvers

The following resolvers are provided and configured by default;

\* `CurrentUserTenantResolveContributor`: Gets the tenant id from claims of the current user, if the current user has logged in. **\*\*This should always be the first contributor for the security\*\***.

\* `QueryStringTenantResolveContributor`: Tries to find current tenant id from query string parameters. The parameter name is `\_\_tenant` by default.

\* `RouteTenantResolveContributor`: Tries to find current tenant id from route (URL path). The variable name is `\_\_tenant` by default. If you defined a route with this variable, then it can determine the current tenant from the route.

\* `HeaderTenantResolveContributor`: Tries to find current tenant id from HTTP headers. The header name is `\_\_tenant` by default.

\* `CookieTenantResolveContributor`: Tries to find current tenant id from cookie values. The cookie name is `\_\_tenant` by default.

###### Problems with the NGINX

You may have problems with the `\_\_tenant` in the HTTP Headers if you're using the [nginx](https://www.nginx.com/) as the reverse proxy server. Because it doesn't allow to use underscore and some other special characters in the HTTP headers and you may need to manually configure it. See the following documents please:

http://nginx.org/en/docs/http/ngx\_http\_core\_module.html#ignore\_invalid\_headers

http://nginx.org/en/docs/http/ngx\_http\_core\_module.html#underscores\_in\_headers

###### AbpAspNetCoreMultiTenancyOptions

`\_\_tenant` parameter name can be changed using `AbpAspNetCoreMultiTenancyOptions`.

**\*\*Example:\*\***

````csharp

services.Configure<AbpAspNetCoreMultiTenancyOptions>(options =>

{

options.TenantKey = "MyTenantKey";

});

````

If you change the `TenantKey`, make sure to pass it to `CoreModule` in the Angular client as follows:

```js

@NgModule({

imports: [

CoreModule.forRoot({

// ...

tenantKey: 'MyTenantKey'

}),

],

// ...

})

export class AppModule {}

```

If you need to access it, you can inject it as follows:

```js

import { Inject } from '@angular/core';

import { TENANT\_KEY } from '@abp/ng.core';

class SomeComponent {

constructor(@Inject(TENANT\_KEY) private tenantKey: string) {}

}

```

> However, we don't suggest to change this value since some clients may assume the the `\_\_tenant` as the parameter name and they might need to manually configure then.

The `MultiTenancyMiddlewareErrorPageBuilder` is used to handle inactive and non-existent tenants.

It will respond to an error page by default, you can change it if you want, eg: only output the error log and continue ASP NET Core's request pipeline.

```csharp

Configure<AbpAspNetCoreMultiTenancyOptions>(options =>

{

options.MultiTenancyMiddlewareErrorPageBuilder = async (context, exception) =>

{

// Handle the exception.

// Return true to stop the pipeline, false to continue.

return true;

};

});

```

##### Domain/Subdomain Tenant Resolver

In a real application, most of times you will want to determine current tenant either by subdomain (like mytenant1.mydomain.com) or by the whole domain (like mytenant.com). If so, you can configure the `AbpTenantResolveOptions` to add the domain tenant resolver.

**\*\*Example: Add a subdomain resolver\*\***

````csharp

Configure<AbpTenantResolveOptions>(options =>

{

options.AddDomainTenantResolver("{0}.mydomain.com");

});

````

\* `{0}` is the placeholder to determine current tenant's unique name.

\* Add this code to the `ConfigureServices` method of your [module](Module-Development-Basics.md).

\* This should be done in the *\*Web/API Layer\** since the URL is a web related stuff.

> There is an [example](https://github.com/abpframework/abp-samples/tree/master/DomainTenantResolver) that uses the subdomain to determining the current tenant.

##### Custom Tenant Resolvers

You can add implement your custom tenant resolver and configure the `AbpTenantResolveOptions` in your module's `ConfigureServices` method as like below:

````csharp

Configure<AbpTenantResolveOptions>(options =>

{

options.TenantResolvers.Add(new MyCustomTenantResolveContributor());

});

````

`MyCustomTenantResolveContributor` must inherit from the `TenantResolveContributorBase` (or implement the `ITenantResolveContributor`) as shown below:

````csharp

using System.Threading.Tasks;

using Volo.Abp.MultiTenancy;

namespace MultiTenancyDemo.Web

{

public class MyCustomTenantResolveContributor : TenantResolveContributorBase

{

public override string Name => "Custom";

public override Task ResolveAsync(ITenantResolveContext context)

{

//TODO...

}

}

}

````

\* A tenant resolver should set `context.TenantIdOrName` if it can determine it. If not, just leave it as is to allow the next resolver to determine it.

\* `context.ServiceProvider` can be used if you need to additional services to resolve from the [dependency injection](Dependency-Injection.md) system.

#### Multi-Tenancy Middleware

Multi-Tenancy middleware is an ASP.NET Core request pipeline [middleware](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware) that determines the current tenant from the HTTP request and sets the `ICurrentTenant` properties.

Multi-Tenancy middleware is typically placed just under the [authentication](https://docs.microsoft.com/en-us/aspnet/core/security/authentication) middleware (`app.UseAuthentication()`):

````csharp

app.UseMultiTenancy();

````

> This middleware is already configured in the startup templates, so you normally don't need to manually add it.

### Tenant Store

`ITenantStore` is used to get the tenant configuration from a data source.

#### Tenant Management Module

The [tenant management module](Modules/Tenant-Management) is **\*\*included in the startup templates\*\*** and implements the `ITenantStore` interface to get the tenants and their configuration from a database. It also provides the necessary functionality and UI to manage the tenants and their connection strings.

#### Configuration Data Store

**\*\*If you don't want to use the tenant management module\*\***, the `DefaultTenantStore` is used as the `ITenantStore` implementation. It gets the tenant configurations from the [configuration system](Configuration.md) (`IConfiguration`). You can either configure the `AbpDefaultTenantStoreOptions` [options](Options.md) or set it in your `appsettings.json` file:

**\*\*Example: Define tenants in appsettings.json\*\***

````json

"Tenants": [

{

"Id": "446a5211-3d72-4339-9adc-845151f8ada0",

"Name": "tenant1"

},

{

"Id": "25388015-ef1c-4355-9c18-f6b6ddbaf89d",

"Name": "tenant2",

"ConnectionStrings": {

"Default": "...tenant2's db connection string here..."

}

}

]

````

> It is recommended to **\*\*use the Tenant Management module\*\***, which is already pre-configured when you create a new application with the ABP startup templates.

### Other Multi-Tenancy Infrastructure

ABP Framework was designed to respect to the multi-tenancy in every aspect and most of the times everything will work as expected.

BLOB Storing, Caching, Data Filtering, Data Seeding, Authorization and all the other services are designed to properly work in a multi-tenant system.

## The Tenant Management Module

ABP Framework provides all the the infrastructure to create a multi-tenant application, but doesn't make any assumption about how you manage (create, delete...) your tenants.

The [Tenant Management module](Modules/Tenant-Management.md) provides a basic UI to manage your tenants and set their connection strings. It is pre-configured for the [application startup template](Startup-Templates/Application.md).

## See Also

\* [Features](Features.md)

## Microservices

# Microservice Architecture

*\*"Microservices are a software development technique—a variant of the* ***\*\*service-oriented architecture\*\**** *(SOA) architectural style that structures an application as a collection of* ***\*\*loosely coupled services\*\*****. In a microservices architecture, services are* ***\*\*fine-grained\*\**** *and the protocols are* ***\*\*lightweight\*\*****. The benefit of decomposing an application into different smaller services is that it improves* ***\*\*modularity\*\*****. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion. It* ***\*\*parallelizes development\*\**** *by enabling small autonomous teams to* ***\*\*develop, deploy and scale\*\**** *their respective services independently. It also allows the architecture of an individual service to emerge through* ***\*\*continuous refactoring\*\*****. Microservices-based architectures enable* ***\*\*continuous delivery and deployment\*\*****."\**

— [Wikipedia](https://en.wikipedia.org/wiki/Microservices)

## Introduction

One of the major goals of the ABP framework is to provide a convenient infrastructure to create microservice solutions. To make this possible,

\* Provides a [module system](Module-Development-Basics.md) that allows you to split your application into modules where each module may have its own database, entities, services, APIs, UI components/pages... etc.

\* Offers an [architectural model](Best-Practices/Module-Architecture.md) to develop your modules to be compatible to microservice development and deployment.

\* Provides [best practices guide](Best-Practices/Index.md) to develop your module standards-compliance.

\* Provides base infrastructure to implement [Domain Driven Design](Domain-Driven-Design.md) in your microservices.

\* Provide services to [automatically create REST-style APIs](API/Auto-API-Controllers.md) from your application services.

\* Provide services to [automatically create C# API clients](API/Dynamic-CSharp-API-Clients.md) that makes easy to consume your services from another service/application.

\* Provides a [distributed event bus](Event-Bus.md) to communicate your services.

\* Provides many other services to make your daily development easier.

## Microservice for New Applications

One common advise to start a new solution is **\*\*always to start with a monolith\*\***, keep it modular and split into microservices once the monolith becomes a problem. This makes your progress fast in the beginning especially if your team is small and you don't want to deal with challenges of the microservice architecture.

However, developing such a well-modular application can be a problem since it is **\*\*hard to keep modules isolated\*\*** from each other as you would do it for microservices (see [Stefan Tilkov's article](https://martinfowler.com/articles/dont-start-monolith.html) about that). Microservice architecture naturally forces you to develop well isolated services, but in a modular monolithic application it's easy to tight couple modules to each other and design **\*\*weak module boundaries\*\*** and API contracts.

ABP can help you in that point by offering a **\*\*microservice-compatible, strict module architecture\*\*** where your module is split into multiple layers/projects and developed in its own VS solution completely isolated and independent from other modules. Such a developed module is a natural microservice yet it can be easily plugged-in a monolithic application. See the [module development best practice guide](Best-Practices/Index.md) that offers a **\*\*microservice-first module design\*\***. All [standard ABP modules](https://github.com/abpframework/abp/tree/master/modules) are developed based on this guide. So, you can use these modules by embedding into your monolithic solution or deploy them separately and use via remote APIs. They can share a single database or can have their own database based on your simple configuration.

## Microservice Demo Solution: eShopOnAbp

The [eShopOnAbp project](https://github.com/abpframework/eShopOnAbp) demonstrates a complete microservice solution based on the ABP framework.

# API

## ABP Endpoints

### Application Configuration

# Application Configuration Endpoint

ABP Framework provides a pre-built and standard endpoint that contains some useful information about the application/service. Here, the list of some fundamental information at this endpoint:

\* Granted [policies](../Authorization.md) (permissions) for the current user.

\* [Setting](../Settings.md) values for the current user.

\* Info about the [current user](../CurrentUser.md) (like id and user name).

\* Info about the current [tenant](../Multi-Tenancy.md) (like id and name).

\* [Time zone](../Timing.md) information for the current user and the [clock](../Timing.md) type of the application.

> If you have started with ABP's startup solution templates and using one of the official UI options, then all these are set up for you and you don't need to know these details. However, if you are building a UI application from scratch, you may want to know this endpoint.

## HTTP API

If you navigate to the `/api/abp/application-configuration` URL of an ABP Framework based web application or HTTP Service, you can access the configuration as a JSON object. This endpoint is useful to create the client of your application.

## Script

For ASP.NET Core MVC (Razor Pages) applications, the same configuration values are also available on the JavaScript side. `/Abp/ApplicationConfigurationScript` is the URL of the script that is auto-generated based on the HTTP API above.

See the [JavaScript API document](../UI/AspNetCore/JavaScript-API/Index.md) for the ASP.NET Core UI.

Other UI types provide services native to the related platform. For example, see the [Angular UI settings documentation](../UI/Angular/Settings.md) to learn how to use the setting values exposes by this endpoint.

### Application Localization

# Application Localization Endpoint

ABP Framework provides a pre-built and standard endpoint that returns all the [localization](../Localization.md) resources and texts defined in the server.

> If you have started with ABP's startup solution templates and using one of the official UI options, then all these are set up for you and you don't need to know these details. However, if you are building a UI application from scratch, you may want to know this endpoint.

## HTTP API

`/api/abp/application-localization` is the main URL of the HTTP API that returns the localization data as a JSON string. I accepts the following query string parameters:

\* `cultureName` (required): A culture code to get the localization data, like `en` or `en-US`.

\* `onlyDynamics` (optional, default: `false`): Can be set to `true` to only get the dynamically defined localization resources and texts. If your client-side application shares the same localization resources with the server (like ABP's Blazor and MVC UIs), you can set `onlyDynamics` to `true`.

**\*\*Example request:\*\***

````

/api/abp/application-localization?cultureName=en

````

## Script

For [ASP.NET Core MVC (Razor Pages)](../UI/AspNetCore/Overall.md) applications, the same localization data is also available on the JavaScript side. `/Abp/ApplicationLocalizationScript` is the URL of the script that is auto-generated based on the HTTP API above.

**\*\*Example request:\*\***

````

/Abp/ApplicationLocalizationScript?cultureName=en

````

See the [JavaScript API document](../UI/AspNetCore/JavaScript-API/Index.md) for the ASP.NET Core UI.

Other UI types provide services native to the related platform. For example, see the [Angular UI localization documentation](../UI/Angular/Localization.md) to learn how to use the localization values exposes by this endpoint.

## API Versioning

# API Versioning System

ABP Framework integrates the [ASPNET-API-Versioning](https://github.com/dotnet/aspnet-api-versioning/wiki) feature and adapts to C# and JavaScript Static Client Proxies and [Auto API Controller](API/Auto-API-Controllers.md).

## Enable API Versioning

```cs

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAbpApiVersioning(options =>

{

// Show neutral/versionless APIs.

options.UseApiBehavior = false;

options.ReportApiVersions = true;

options.AssumeDefaultVersionWhenUnspecified = true;

});

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ChangeControllerModelApiExplorerGroupName = false;

});

}

```

## C# and JavaScript Static Client Proxies

This feature does not compatible with [URL Path Versioning](https://github.com/dotnet/aspnet-api-versioning/wiki/Versioning-via-the-URL-Path), we suggest to use [Versioning-via-the-Query-String](https://github.com/dotnet/aspnet-api-versioning/wiki/Versioning-via-the-Query-String).

### Example

**\*\*Application Services:\*\***

```cs

public interface IBookAppService : IApplicationService

{

Task<BookDto> GetAsync();

}

public interface IBookV2AppService : IApplicationService

{

Task<BookDto> GetAsync();

Task<BookDto> GetAsync(string isbn);

}

```

**\*\*HttpApi Controllers:\*\***

```cs

[Area(BookStoreRemoteServiceConsts.ModuleName)]

[RemoteService(Name = BookStoreRemoteServiceConsts.RemoteServiceName)]

[ApiVersion("1.0", Deprecated = true)]

[ApiController]

[ControllerName("Book")]

[Route("api/BookStore/Book")]

public class BookController : BookStoreController, IBookAppService

{

private readonly IBookAppService \_bookAppService;

public BookController(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

[HttpGet]

public async Task<BookDto> GetAsync()

{

return await \_bookAppService.GetAsync();

}

}

[Area(BookStoreRemoteServiceConsts.ModuleName)]

[RemoteService(Name = BookStoreRemoteServiceConsts.RemoteServiceName)]

[ApiVersion("2.0")]

[ApiController]

[ControllerName("Book")]

[Route("api/BookStore/Book")]

public class BookV2Controller : BookStoreController, IBookV2AppService

{

private readonly IBookV2AppService \_bookAppService;

public BookV2Controller(IBookV2AppService bookAppService)

{

\_bookAppService = bookAppService;

}

[HttpGet]

public async Task<BookDto> GetAsync()

{

return await \_bookAppService.GetAsync();

}

[HttpGet]

[Route("{isbn}")]

public async Task<BookDto> GetAsync(string isbn)

{

return await \_bookAppService.GetAsync(isbn);

}

}

```

**\*\*Generated CS and JS proxies:\*\***

```cs

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IBookAppService), typeof(BookClientProxy))]

public partial class BookClientProxy : ClientProxyBase<IBookAppService>, IBookAppService

{

public virtual async Task<BookDto> GetAsync()

{

return await RequestAsync<BookDto>(nameof(GetAsync));

}

}

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IBookV2AppService), typeof(BookV2ClientProxy))]

public partial class BookV2ClientProxy : ClientProxyBase<IBookV2AppService>, IBookV2AppService

{

public virtual async Task<BookDto> GetAsync()

{

return await RequestAsync<BookDto>(nameof(GetAsync));

}

public virtual async Task<BookDto> GetAsync(string isbn)

{

return await RequestAsync<BookDto>(nameof(GetAsync), new ClientProxyRequestTypeValue

{

{ typeof(string), isbn }

});

}

}

```

```js

// controller bookStore.books.book

(function(){

abp.utils.createNamespace(window, 'bookStore.books.book');

bookStore.books.book.get = function(api\_version, ajaxParams) {

var api\_version = api\_version ? api\_version : '1.0';

return abp.ajax($.extend(true, {

url: abp.appPath + 'api/BookStore/Book' + abp.utils.buildQueryString([{ name: 'api-version', value: api\_version }]) + '',

type: 'GET'

}, ajaxParams));

};

})();

// controller bookStore.books.bookV2

(function(){

abp.utils.createNamespace(window, 'bookStore.books.bookV2');

bookStore.books.bookV2.get = function(api\_version, ajaxParams) {

var api\_version = api\_version ? api\_version : '2.0';

return abp.ajax($.extend(true, {

url: abp.appPath + 'api/BookStore/Book' + abp.utils.buildQueryString([{ name: 'api-version', value: api\_version }]) + '',

type: 'GET'

}, ajaxParams));

};

bookStore.books.bookV2.getAsyncByIsbn = function(isbn, api\_version, ajaxParams) {

var api\_version = api\_version ? api\_version : '2.0';

return abp.ajax($.extend(true, {

url: abp.appPath + 'api/BookStore/Book/' + isbn + '' + abp.utils.buildQueryString([{ name: 'api-version', value: api\_version }]) + '',

type: 'GET'

}, ajaxParams));

};

})();

```

## Changing version manually

If an application service class supports multiple versions. You can inject `ICurrentApiVersionInfo` to switch versions in C#.

```cs

var currentApiVersionInfo = \_abpApplication.ServiceProvider.GetRequiredService<ICurrentApiVersionInfo>();

var bookV4AppService = \_abpApplication.ServiceProvider.GetRequiredService<IBookV4AppService>();

using (currentApiVersionInfo.Change(new ApiVersionInfo(ParameterBindingSources.Query, "4.0")))

{

book = await bookV4AppService.GetAsync();

logger.LogWarning(book.Title);

logger.LogWarning(book.ISBN);

}

using (currentApiVersionInfo.Change(new ApiVersionInfo(ParameterBindingSources.Query, "4.1")))

{

book = await bookV4AppService.GetAsync();

logger.LogWarning(book.Title);

logger.LogWarning(book.ISBN);

}

```

We have made a default version in the JS proxy. Of course, you can also manually change the version.

```js

bookStore.books.bookV4.get("4.0") // Manually change the version.

//Title: Mastering ABP Framework V4.0

bookStore.books.bookV4.get() // The latest supported version is used by default.

//Title: Mastering ABP Framework V4.1

```

## Auto API Controller

```cs

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<AbpAspNetCoreMvcOptions>(options =>

{

//2.0 Version

options.ConventionalControllers.Create(typeof(BookStoreWebAppModule).Assembly, opts =>

{

opts.TypePredicate = t => t.Namespace == typeof(BookStore.Controllers.ConventionalControllers.v2.TodoAppService).Namespace;

opts.ApiVersions.Add(new ApiVersion(2, 0));

});

//1.0 Compatibility version

options.ConventionalControllers.Create(typeof(BookStoreWebAppModule).Assembly, opts =>

{

opts.TypePredicate = t => t.Namespace == typeof(BookStore.Controllers.ConventionalControllers.v1.TodoAppService).Namespace;

opts.ApiVersions.Add(new ApiVersion(1, 0));

});

});

}

public override void ConfigureServices(ServiceConfigurationContext context)

{

var preActions = context.Services.GetPreConfigureActions<AbpAspNetCoreMvcOptions>();

Configure<AbpAspNetCoreMvcOptions>(options =>

{

preActions.Configure(options);

});

context.Services.AddAbpApiVersioning(options =>

{

// Show neutral/versionless APIs.

options.UseApiBehavior = false;

options.ReportApiVersions = true;

options.AssumeDefaultVersionWhenUnspecified = true;

options.ConfigureAbp(preActions.Configure());

});

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ChangeControllerModelApiExplorerGroupName = false;

});

}

```

## Swagger/VersionedApiExplorer

```cs

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAbpApiVersioning(options =>

{

// Show neutral/versionless APIs.

options.UseApiBehavior = false;

options.ReportApiVersions = true;

options.AssumeDefaultVersionWhenUnspecified = true;

});

context.Services.AddVersionedApiExplorer(

options =>

{

// add the versioned api explorer, which also adds IApiVersionDescriptionProvider service

// note: the specified format code will format the version as "'v'major[.minor][-status]"

options.GroupNameFormat = "'v'VVV";

// note: this option is only necessary when versioning by url segment. the SubstitutionFormat

// can also be used to control the format of the API version in route templates

options.SubstituteApiVersionInUrl = true;

});

context.Services.AddTransient<IConfigureOptions<SwaggerGenOptions>, ConfigureSwaggerOptions>();

context.Services.AddAbpSwaggerGen(

options =>

{

// add a custom operation filter which sets default values

options.OperationFilter<SwaggerDefaultValues>();

options.CustomSchemaIds(type => type.FullName);

});

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ChangeControllerModelApiExplorerGroupName = false;

});

}

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

var env = context.GetEnvironment();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseErrorPage();

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAbpRequestLocalization();

app.UseSwagger();

app.UseSwaggerUI(

options =>

{

var provider = app.ApplicationServices.GetRequiredService<IApiVersionDescriptionProvider>();

// build a swagger endpoint for each discovered API version

foreach (var description in provider.ApiVersionDescriptions)

{

options.SwaggerEndpoint($"/swagger/{description.GroupName}/swagger.json", description.GroupName.ToUpperInvariant());

}

});

app.UseConfiguredEndpoints();

}

```

## Custom multi-version API controller

ABP Framework will not affect to your APIs, you can freely implement your APIs according to the Microsoft's documentation.

Further information, see https://github.com/dotnet/aspnet-api-versioning/wiki

## Sample source code

Follow the link below to get the sample's complete source-code

https://github.com/abpframework/abp-samples/tree/master/Api-Versioning

## Auto API Controllers

# Auto API Controllers

Once you create an [application service](../Application-Services.md), you generally want to create an API controller to expose this service as an HTTP (REST) API endpoint. A typical API controller does nothing but redirects method calls to the application service and configures the REST API using attributes like [HttpGet], [HttpPost], [Route]... etc.

ABP can **\*\*automagically\*\*** configure your application services as API Controllers by convention. Most of time you don't care about its detailed configuration, but it's possible to fully customize it.

## Configuration

Basic configuration is simple. Just configure `AbpAspNetCoreMvcOptions` and use `ConventionalControllers.Create` method as shown below:

````csharp

[DependsOn(BookStoreApplicationModule)]

public class BookStoreWebModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options

.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly);

});

}

}

````

This example code configures all the application services in the assembly containing the class `BookStoreApplicationModule`. The figure below shows the resulting API on the [Swagger UI](https://swagger.io/tools/swagger-ui/).

![bookstore-apis](../images/bookstore-apis.png)

### Examples

Some example method names and the corresponding routes calculated by convention:

| Service Method Name | HTTP Method | Route |

| ----------------------------------------------------- | ----------- | -------------------------- |

| GetAsync(Guid id) | GET | /api/app/book/{id} |

| GetListAsync() | GET | /api/app/book |

| CreateAsync(CreateBookDto input) | POST | /api/app/book |

| UpdateAsync(Guid id, UpdateBookDto input) | PUT | /api/app/book/{id} |

| DeleteAsync(Guid id) | DELETE | /api/app/book/{id} |

| GetEditorsAsync(Guid id) | GET | /api/app/book/{id}/editors |

| CreateEditorAsync(Guid id, BookEditorCreateDto input) | POST | /api/app/book/{id}/editor |

### HTTP Method

ABP uses a naming convention while determining the HTTP method for a service method (action):

- **\*\*Get\*\***: Used if the method name starts with 'GetList', 'GetAll' or 'Get'.

- **\*\*Put\*\***: Used if the method name starts with 'Put' or 'Update'.

- **\*\*Delete\*\***: Used if the method name starts with 'Delete' or 'Remove'.

- **\*\*Post\*\***: Used if the method name starts with 'Create', 'Add', 'Insert' or 'Post'.

- **\*\*Patch\*\***: Used if the method name starts with 'Patch'.

- Otherwise, **\*\*Post\*\*** is used **\*\*by default\*\***.

If you need to customize HTTP method for a particular method, then you can use one of the standard ASP.NET Core attributes ([HttpPost], [HttpGet], [HttpPut]... etc.). This requires to add [Microsoft.AspNetCore.Mvc.Core](https://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Core) nuget package to your project that contains the service.

### Route

Route is calculated based on some conventions:

\* It always starts with '**\*\*/api\*\***'.

\* Continues with a **\*\*route path\*\***. Default value is '**\*\*/app\*\***' and can be configured as like below:

````csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.RootPath = "volosoft/book-store";

});

});

````

Then the route for getting a book will be '**\*\*/api/volosoft/book-store/book/{id}\*\***'. This sample uses two-level root path, but you generally use a single level depth.

\* Continues with the **\*\*normalized controller/service name\*\***. Normalization removes 'AppService', 'ApplicationService' and 'Service' postfixes and converts it to **\*\*kebab-case\*\***. If your application service class name is 'ReadingBookAppService' then it becomes only '/reading-book'.

\* If you want to customize naming, then set the `UrlControllerNameNormalizer` option. It's a func delegate which allows you to determine the name per controller/service.

\* If the method has an '**\*\*id\*\***' parameter then it adds '**\*\*/{id}\*\***' ro the route.

\* Then it adds the action name if necessary. Action name is obtained from the method name on the service and normalized by;

\* Removing '**\*\*Async\*\***' postfix. If the method name is 'GetPhonesAsync' then it becomes 'GetPhones'.

\* Removing **\*\*HTTP method prefix\*\***. 'GetList', 'GetAll', 'Get', 'Put', 'Update', 'Delete', 'Remove', 'Create', 'Add', 'Insert', 'Post' and 'Patch' prefixes are removed based on the selected HTTP method. So, 'GetPhones' becomes 'Phones' since 'Get' prefix is a duplicate for a GET request.

\* Converting the result to **\*\*kebab-case\*\***.

\* If the resulting action name is **\*\*empty\*\*** then it's not added to the route. If it's not empty, it's added to the route (like '/phones'). For 'GetAllAsync' method name it will be empty, for 'GetPhonesAsync' method name it will be 'phones'.

\* Normalization can be customized by setting the `UrlActionNameNormalizer` option. It's an action delegate that is called for every method.

\* If there is another parameter with 'Id' postfix, then it's also added to the route as the final route segment (like '/phoneId').

#### Customizing the Route Calculation

`IConventionalRouteBuilder` is used to build the route. It is implemented by the `ConventionalRouteBuilder` by default and works as explained above. You can replace/override this service to customize the route calculation strategy.

#### Version 3.x Style Route Calculation

The route calculation was different before the version 4.0. It was using camelCase conventions, while the ABP Framework version 4.0+ uses kebab-case. If you use the old route calculation strategy, follow one of the approaches;

\* Set `UseV3UrlStyle` to `true` in the options of the `options.ConventionalControllers.Create(...)` method. Example:

````csharp

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.UseV3UrlStyle = true;

});

````

This approach effects only the controllers for the `BookStoreApplicationModule`.

\* Set `UseV3UrlStyle` to `true` for the `AbpConventionalControllerOptions` to set it globally. Example:

```csharp

Configure<AbpConventionalControllerOptions>(options =>

{

options.UseV3UrlStyle = true;

});

```

Setting it globally effects all the modules in a modular application.

## Service Selection

Creating conventional HTTP API controllers are not unique to application services actually.

### IRemoteService Interface

If a class implements the `IRemoteService` interface then it's automatically selected to be a conventional API controller. Since application services inherently implement it, they are considered as natural API controllers.

### RemoteService Attribute

`RemoteService` attribute can be used to mark a class as a remote service or disable for a particular class that inherently implements the `IRemoteService` interface. Example:

````csharp

[RemoteService(IsEnabled = false)] //or simply [RemoteService(false)]

public class PersonAppService : ApplicationService

{

}

````

### TypePredicate Option

You can further filter classes to become an API controller by providing the `TypePredicate` option:

````csharp

services.Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.TypePredicate = type => { return true; };

});

});

````

Instead of returning `true` for every type, you can check it and return `false` if you don't want to expose this type as an API controller.

## API Explorer

API Exploring a service that makes possible to investigate API structure by the clients. Swagger uses it to create a documentation and test UI for an endpoint.

API Explorer is automatically enabled for conventional HTTP API controllers by default. Use `RemoteService` attribute to control it per class or method level. Example:

````csharp

[RemoteService(IsMetadataEnabled = false)]

public class PersonAppService : ApplicationService

{

}

````

Disabled `IsMetadataEnabled` which hides this service from API explorer and it will not be discoverable. However, it still can be usable for the clients know the exact API path/route.

## Replace or Remove Controllers.

In addition to [Overriding a Controller](../Customizing-Application-Modules-Overriding-Services.md#example-overriding-a-controller), you can also use a completely independent **\*\*Controller\*\*** to replace the controller in the framework or module.

They have the same [route](https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-7.0), but can have **\*\*different\*\*** input and output parameters.

### Replace built-in AbpApplicationConfigurationController

The `ReplaceControllersAttribute` indicates the replaced controller type.

````csharp

[ReplaceControllers(typeof(AbpApplicationConfigurationController))]

[Area("abp")]

[RemoteService(Name = "abp")]

public class ReplaceBuiltInController : AbpController

{

[HttpGet("api/abp/application-configuration")]

public virtual Task<MyApplicationConfigurationDto> GetAsync(MyApplicationConfigurationRequestOptions options)

{

return Task.FromResult(new MyApplicationConfigurationDto());

}

}

public class MyApplicationConfigurationRequestOptions : ApplicationConfigurationRequestOptions

{

}

public class MyApplicationConfigurationDto : ApplicationConfigurationDto

{

}

````

### Remove controller

Configure `ControllersToRemove` of `AbpAspNetCoreMvcOptions` to remove the controllers.

````csharp

services.Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ControllersToRemove.Add(typeof(AbpLanguagesController));

});

````

## Dynamic C# API Clients

# Dynamic C# API Client Proxies

ABP can dynamically create C# API client proxies to call your remote HTTP services (REST APIs). In this way, you don't need to deal with `HttpClient` and other low level details to call remote services and get results.

Dynamic C# proxies automatically handle the following stuff for you;

\* Maps C# **\*\*method calls\*\*** to remote server **\*\*HTTP calls\*\*** by considering the HTTP method, route, query string parameters, request payload and other details.

\* **\*\*Authenticates\*\*** the HTTP Client by adding access token to the HTTP header.

\* **\*\*Serializes\*\*** to and deserialize from JSON.

\* Handles HTTP API **\*\*versioning\*\***.

\* Add **\*\*correlation id\*\***, current **\*\*tenant\*\*** id and the current **\*\*culture\*\*** to the request.

\* Properly **\*\*handles the error messages\*\*** sent by the server and throws proper exceptions.

This system can be used by any type of .NET client to consume your HTTP APIs.

## Static vs Dynamic Client Proxies

ABP provides **\*\*two types\*\*** of client proxy generation system. This document explains the **\*\*dynamic client proxies\*\***, which generates client-side proxies on runtime. You can also see the [Static C# API Client Proxies](Static-CSharp-API-Clients.md) documentation to learn how to generate proxies on development time.

Development-time (static) client proxy generation has a **\*\*performance advantage\*\*** since it doesn't need to obtain the HTTP API definition on runtime. However, you should **\*\*re-generate\*\*** the client proxy code whenever you change your API endpoint definition. On the other hand, dynamic client proxies are generated on runtime and provides an **\*\*easier development experience\*\***.

## Service Interface

Your service/controller should implement an interface that is shared between the server and the client. So, first define a service interface in a shared library project, typically in the `Application.Contracts` project if you've created your solution using the startup templates.

Example:

````csharp

public interface IBookAppService : IApplicationService

{

Task<List<BookDto>> GetListAsync();

}

````

> Your interface should implement the `IRemoteService` interface to be automatically discovered. Since the `IApplicationService` inherits the `IRemoteService` interface, the `IBookAppService` above satisfies this condition.

Implement this class in your service application. You can use [auto API controller system](Auto-API-Controllers.md) to expose the service as a REST API endpoint.

## Client Proxy Generation

> The startup templates already comes pre-configured for the client proxy generation, in the `HttpApi.Client` project.

If you're not using a startup template, then execute the following command in the folder that contains the .csproj file of your client project:

````

abp add-package Volo.Abp.Http.Client

````

> If you haven't done it yet, you first need to install the [ABP CLI](../CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Http.Client).

Now, it's ready to create the client proxies. Example:

````csharp

[DependsOn(

typeof(AbpHttpClientModule), //used to create client proxies

typeof(BookStoreApplicationContractsModule) //contains the application service interfaces

)]

public class MyClientAppModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Create dynamic client proxies

context.Services.AddHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly

);

}

}

````

`AddHttpClientProxies` method gets an assembly, finds all service interfaces in the given assembly, creates and registers proxy classes.

### Endpoint Configuration

`RemoteServices` section in the `appsettings.json` file is used to get remote service address by default. The simplest configuration is shown below:

```json

{

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:53929/"

}

}

}

```

See the "AbpRemoteServiceOptions" section below for more detailed configuration.

## Usage

It's straightforward to use. Just inject the service interface in the client application code:

````csharp

public class MyService : ITransientDependency

{

private readonly IBookAppService \_bookService;

public MyService(IBookAppService bookService)

{

\_bookService = bookService;

}

public async Task DoIt()

{

var books = await \_bookService.GetListAsync();

foreach (var book in books)

{

Console.WriteLine($"[BOOK {book.Id}] Name={book.Name}");

}

}

}

````

This sample injects the `IBookAppService` service interface defined above. The dynamic client proxy implementation makes an HTTP call whenever a service method is called by the client.

### IHttpClientProxy Interface

While you can inject `IBookAppService` like above to use the client proxy, you could inject `IHttpClientProxy<IBookAppService>` for a more explicit usage. In this case you will use the `Service` property of the `IHttpClientProxy<T>` interface.

## Configuration

### AbpRemoteServiceOptions

`AbpRemoteServiceOptions` is automatically set from the `appsettings.json` by default. Alternatively, you can configure it in the `ConfigureServices` method of your [module](../Module-Development-Basics.md) to set or override it. Example:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.Configure<AbpRemoteServiceOptions>(options =>

{

options.RemoteServices.Default =

new RemoteServiceConfiguration("http://localhost:53929/");

});

//...

}

````

### Multiple Remote Service Endpoints

The examples above have configured the "Default" remote service endpoint. You may have different endpoints for different services (as like in a microservice approach where each microservice has different endpoints). In this case, you can add other endpoints to your configuration file:

````json

{

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:53929/"

},

"BookStore": {

"BaseUrl": "http://localhost:48392/"

}

}

}

````

`AddHttpClientProxies` method can get an additional parameter for the remote service name. Example:

````csharp

context.Services.AddHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly,

remoteServiceConfigurationName: "BookStore"

);

````

`remoteServiceConfigurationName` parameter matches the service endpoint configured via `AbpRemoteServiceOptions`. If the `BookStore` endpoint is not defined then it fallbacks to the `Default` endpoint.

### As Default Services

When you create a service proxy for `IBookAppService`, you can directly inject the `IBookAppService` to use the proxy client (as shown in the usage section). You can pass `asDefaultServices: false` to the `AddHttpClientProxies` method to disable this feature.

````csharp

context.Services.AddHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly,

asDefaultServices: false

);

````

Using `asDefaultServices: false` may only be needed if your application has already an implementation of the service and you do not want to override/replace the other implementation by your client proxy.

> If you disable `asDefaultServices`, you can only use `IHttpClientProxy<T>` interface to use the client proxies. See the *\*IHttpClientProxy Interface\** section above.

### Retry/Failure Logic & Polly Integration

If you want to add retry logic for the failing remote HTTP calls for the client proxies, you can configure the `AbpHttpClientBuilderOptions` in the `PreConfigureServices` method of your module class.

**\*\*Example: Use the [**Polly**](https://github.com/App-vNext/Polly) library to re-try 3 times on a failure\*\***

````csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<AbpHttpClientBuilderOptions>(options =>

{

options.ProxyClientBuildActions.Add((remoteServiceName, clientBuilder) =>

{

clientBuilder.AddTransientHttpErrorPolicy(policyBuilder =>

policyBuilder.WaitAndRetryAsync(

3,

i => TimeSpan.FromSeconds(Math.Pow(2, i))

)

);

});

});

}

````

This example uses the [Microsoft.Extensions.Http.Polly](https://www.nuget.org/packages/Microsoft.Extensions.Http.Polly) package. You also need to import the `Polly` namespace (`using Polly;`) to be able to use the `WaitAndRetryAsync` method.

## See Also

\* [Static C# Client Proxies](Static-CSharp-API-Clients.md)

## Integration Services

# Integration Services

The *\*Integration Service\** concept was created to distinguish the [application services](Application-Services.md) that are built for inter-module (or inter-microservice) communication from the application services that are intended to be consumed from a UI or a client application.

The following figure shows a few microservices behind an API Gateway that is consumed by a UI application and 3rd-party client applications:

![integration-services](images/integration-services.png)

HTTP requests coming from out of the API Gateway can be called as *\*external request\**, while the HTTP requests performed between microservices can be considered as *\*internal requests\**. The application services that are designed to respond to these internal requests are called as *\*integration services\**, because their purpose is to integrate microservices in the system, rather than respond to user requests.

## Marking an Application Service as Integration Service

Assume that you have an application service named `ProductAppService`, and you want to use that application service as an integration service. In that case, you can use the `[IntegrationService]` attribute on top of the application service class as shown below:

```csharp

[IntegrationService]

public class ProductAppService : ApplicationService, IProductAppService

{

// ...

}

```

If your application service has an interface, like `IProductService` in this example, you can use it on the service interface:

```csharp

[IntegrationService]

public interface IProductAppService : IApplicationService

{

// ...

}

```

> If you've used the `[IntegrationService]` on top of your service interface, it is *\*not needed\** to use on the service class too.

That's all. From now, ABP will handle your application service as integration service and implement the followings by convention:

\* That service is **\*\*not exposed\*\*** by default, unless you explicitly set `ExposeIntegrationServices` options (see the *\*Exposing Integration Services\** section).

\* If you are using the [Auto API Controllers](API/Auto-API-Controllers.md) feature in your application, the **\*\*URL prefix\*\*** will be `/integration-api` instead of `/api` for your integration services. Thus, you can distinguish internal and external service communications and take additional actions, such as preventing REST API calls for integration services out of API Gateway.

\* **\*\*Audit logging\*\*** is disabled by default for the integration services. See the next section if you want to enable it.

## Marking an MVC Controller as Integration Service

In addition to application services, you can mark a regular MVC Controller as integration service, using the same `IntegrationService` attribute, or inheriting an interface that has the `IntegrationService` attribute.

**\*\*Example:\*\***

````csharp

[IntegrationService] // Mark as integration service

[Route("integration-api/products")]

public class ProductController : AbpControllerBase

{

//...

}

````

When you use the `IntegrationService` attribute, ABP will handle your controller as integration service and implement the followings by convention:

\* That controller is **\*\*not exposed\*\*** to clients by default, unless you explicitly set `ExposeIntegrationServices` options (see the *\*Exposing Integration Services\** section).

\* **\*\*Audit logging\*\*** is disabled by default for controller. See the next section if you want to enable it.

## Configuration

### Exposing Integration Services

Integration services and controllers are not exposed by default for security reasons. They typically don't require authorization, so you should **\*\*carefully and explicitly\*\*** allow them to be visible and usable to client applications.

To expose integration services and controllers, set `AbpAspNetCoreMvcOptions.ExposeIntegrationServices` to `true` in the `ConfigureServices` method of your [module class](Module-Development-Basics.md):

````csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ExposeIntegrationServices = true;

});

````

> Hiding integration services is useful when you are building reusable application modules, where they may be used in a monolith application or in a microservice system. In a monolith application, integration services don't need to be exposed outside since the modules may in-process communicate with each other. On the other hand, if you build a microservice solution and use that module as a service, it will be proper to expose the integration services, so other microservices can consume them remotely inside your private network (or Kubernetes cluster). In that case, be careful to not accidently expose the integration services out of your private network. Configuring your API Gateway so that it blocks requests to `integration-api` prefixed URLs from outside of your network will be a good option.

### Enabling/Disabling the Audit Logging

Audit Logging is disabled by default for integration services but it can be enabled by configuring the `AbpAuditingOptions` [options class](Options.md) in the `ConfigureServices` method of your [module class](Module-Development-Basics.md):

```csharp

Configure<AbpAuditingOptions>(options =>

{

options.IsEnabledForIntegrationService = true;

});

```

> Please refer to the [audit logging document](Audit-Logging.md) for other options and details.

### Filtering Auto API Controllers

You can filter integration services (or non-integration services) while creating [Auto API Controllers](API/Auto-API-Controllers.md), using the `ApplicationServiceTypes` option of the `ConventionalControllerSetting` by configuring the `AbpAspNetCoreMvcOptions` as shown below:

```csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers.Create(

typeof(MyApplicationModule).Assembly,

conventionalControllerSetting =>

{

conventionalControllerSetting.ApplicationServiceTypes =

ApplicationServiceTypes.IntegrationServices;

});

});

```

Tip: You can call the `options.ConventionalControllers.Create` multiple times to configure regular application services and integration services with different options.

> Please refer to the [Auto API Controllers document](API/Auto-API-Controllers.md) for more information about the Auto API Controller system.

## See Also

\* [Application Services](Application-Services.md)

\* [Auto API Controllers](API/Auto-API-Controllers.md)

\* [Audit Logging](Audit-Logging.md)

## Static C# API Clients

# Static C# API Client Proxies

ABP can create C# API client proxy code to call your remote HTTP services (REST APIs). In this way, you don't need to deal with `HttpClient` and other low level details to call remote services and get results.

Static C# proxies automatically handle the following stuff for you;

\* Maps C# **\*\*method calls\*\*** to remote server **\*\*HTTP calls\*\*** by considering the HTTP method, route, query string parameters, request payload and other details.

\* **\*\*Authenticates\*\*** the HTTP Client by adding access token to the HTTP header.

\* **\*\*Serializes\*\*** to and deserialize from JSON.

\* Handles HTTP API **\*\*versioning\*\***.

\* Add **\*\*correlation id\*\***, current **\*\*tenant\*\*** id and the current **\*\*culture\*\*** to the request.

\* Properly **\*\*handles the error messages\*\*** sent by the server and throws proper exceptions.

This system can be used by any type of .NET client to consume your HTTP APIs.

## Static vs Dynamic Client Proxies

ABP provides **\*\*two types\*\*** of client proxy generation system. This document explains the **\*\*static client proxies\*\***, which generates client-side code in your development time. You can also see the [Dynamic C# API Client Proxies](Dynamic-CSharp-API-Clients.md) documentation to learn how to use proxies generated on runtime.

Development-time (static) client proxy generation has a **\*\*performance advantage\*\*** since it doesn't need to obtain the HTTP API definition on runtime. However, you should **\*\*re-generate\*\*** the client proxy code whenever you change your API endpoint definition. On the other hand, dynamic client proxies are generated on runtime and provides an **\*\*easier development experience\*\***.

## Service Interface

Your service/controller should implement an interface that is shared between the server and the client. So, first define a service interface in a shared library project, typically in the `Application.Contracts` project if you've created your solution using the startup templates.

Example:

````csharp

public interface IBookAppService : IApplicationService

{

Task<List<BookDto>> GetListAsync();

}

````

> Your interface should implement the `IRemoteService` interface to be automatically discovered. Since the `IApplicationService` inherits the `IRemoteService` interface, the `IBookAppService` above satisfies this condition.

Implement this class in your service application. You can use [auto API controller system](Auto-API-Controllers.md) to expose the service as a REST API endpoint.

## With Contracts or Without Contracts

`Without Contracts` depending on target service's `application.contracts` package, so they can reuse the DTOs and other related classes. However, that can be a problem when we want to create fully independently developed and deployed microservices. We want to use the static proxy generation even without depending target service's application.contracts package.

`With Contracts` generate all the `classes/enums/other` types in the client side (including application service interfaces) , This is also the default behavior of the `generate-proxy` command.

## Client Proxy Generation

First, add [Volo.Abp.Http.Client](https://www.nuget.org/packages/Volo.Abp.Http.Client) nuget package to your client project:

````

Install-Package Volo.Abp.Http.Client

````

Then add `AbpHttpClientModule` dependency to your module:

````csharp

[DependsOn(typeof(AbpHttpClientModule))] //add the dependency

public class MyClientAppModule : AbpModule

{

}

````

Now, it's ready to configure the application for the static client proxy generation.

### With Contracts Example

````csharp

[DependsOn(

typeof(AbpHttpClientModule) //used to create client proxies

)]

public class MyClientAppModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

// Prepare for static client proxy generation

context.Services.AddStaticHttpClientProxies(

typeof(MyClientAppModule).Assembly

);

}

}

````

### Without Contracts Example

````csharp

[DependsOn(

typeof(AbpHttpClientModule), //used to create client proxies

typeof(BookStoreApplicationContractsModule) //contains the application service interfaces

)]

public class MyClientAppModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

// Prepare for static client proxy generation

context.Services.AddStaticHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly

);

}

}

````

`AddStaticHttpClientProxies` method gets an assembly, finds all service interfaces in the given assembly, and prepares for static client proxy generation.

> The [application startup template](../Startup-Templates/Application.md) comes pre-configured for the **\*\*dynamic\*\*** client proxy generation, in the `HttpApi.Client` project. If you want to switch to the **\*\*static\*\*** client proxies, change `context.Services.AddHttpClientProxies` to `context.Services.AddStaticHttpClientProxies` in the module class of your `HttpApi.Client` project.

### Endpoint Configuration

`RemoteServices` section in the `appsettings.json` file is used to get remote service address by default. The simplest configuration is shown below:

```json

{

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:53929/"

}

}

}

```

See the *\*AbpRemoteServiceOptions\** section below for more detailed configuration.

### Code Generation

Server side must be up and running while generating the client proxy code. So, run your application that serves the HTTP APIs on the `BaseUrl` that is configured like explained in the *\*Endpoint Configuration\** section.

Open a command-line terminal in the root folder of your client project (`.csproj`) and type the following command:

#### With Contracts

````bash

abp generate-proxy -t csharp -u http://localhost:53929/

````

> If you haven't installed yet, you should install the [ABP CLI](../CLI.md).

This command should generate the following files under the `ClientProxies` folder:

![generated-static-client-proxies](../images/generated-static-client-proxies-with-contracts.png)

\* `BookClientProxy.Generated.cs` is the actual generated proxy class in this example. `BookClientProxy` is a `partial` class \* where you can write your custom code (ABP won't override it).

\* `IBookAppService.cs` is the app service.

\* `BookDto.cs` is the Dto class which uses by app service.

\* `app-generate-proxy.json` contains information about the remote HTTP endpoint, so ABP can properly perform HTTP requests.

#### Without Contracts

````bash

abp generate-proxy -t csharp -u http://localhost:53929/ --without-contracts

````

This command should generate the following files under the `ClientProxies` folder:

![generated-static-client-proxies](../images/generated-static-client-proxies-without-contracts.png)

\* `BookClientProxy.Generated.cs` is the actual generated proxy class in this example. `BookClientProxy` is a `partial` class where you can write your custom code (ABP won't override it).

\* `app-generate-proxy.json` contains information about the remote HTTP endpoint, so ABP can properly perform HTTP requests.

> `generate-proxy` command generates proxies for only the APIs you've defined in your application. If you are developing a modular application, you can specify the `-m` (or `--module`) parameter to specify the module you want to generate proxies. See the *\*generate-proxy\** section in the [ABP CLI](../CLI.md) documentation for other options.

## Usage

It's straightforward to use the client proxies. Just inject the service interface in the client application code:

````csharp

public class MyService : ITransientDependency

{

private readonly IBookAppService \_bookService;

public MyService(IBookAppService bookService)

{

\_bookService = bookService;

}

public async Task DoItAsync()

{

var books = await \_bookService.GetListAsync();

foreach (var book in books)

{

Console.WriteLine($"[BOOK {book.Id}] Name={book.Name}");

}

}

}

````

This sample injects the `IBookAppService` service interface defined above. The static client proxy implementation makes an HTTP call whenever a service method is called by the client.

## Configuration

### AbpRemoteServiceOptions

`AbpRemoteServiceOptions` is automatically set from the `appsettings.json` by default. Alternatively, you can configure it in the `ConfigureServices` method of your [module](../Module-Development-Basics.md) to set or override it. Example:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.Configure<AbpRemoteServiceOptions>(options =>

{

options.RemoteServices.Default =

new RemoteServiceConfiguration("http://localhost:53929/");

});

//...

}

````

### Multiple Remote Service Endpoints

The examples above have configured the "Default" remote service endpoint. You may have different endpoints for different services (as like in a microservice approach where each microservice has different endpoints). In this case, you can add other endpoints to your configuration file:

````json

{

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:53929/"

},

"BookStore": {

"BaseUrl": "http://localhost:48392/"

}

}

}

````

`AddStaticHttpClientProxies` method can get an additional parameter for the remote service name. Example:

````csharp

context.Services.AddStaticHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly,

remoteServiceConfigurationName: "BookStore"

);

````

`remoteServiceConfigurationName` parameter matches the service endpoint configured via `AbpRemoteServiceOptions`. If the `BookStore` endpoint is not defined then it fallbacks to the `Default` endpoint.

### Retry/Failure Logic & Polly Integration

If you want to add retry logic for the failing remote HTTP calls for the client proxies, you can configure the `AbpHttpClientBuilderOptions` in the `PreConfigureServices` method of your module class.

**\*\*Example: Use the [**Polly**](https://github.com/App-vNext/Polly) library to re-try 3 times on a failure\*\***

````csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<AbpHttpClientBuilderOptions>(options =>

{

options.ProxyClientBuildActions.Add((remoteServiceName, clientBuilder) =>

{

clientBuilder.AddTransientHttpErrorPolicy(policyBuilder =>

policyBuilder.WaitAndRetryAsync(

3,

i => TimeSpan.FromSeconds(Math.Pow(2, i))

)

);

});

});

}

````

This example uses the [Microsoft.Extensions.Http.Polly](https://www.nuget.org/packages/Microsoft.Extensions.Http.Polly) package. You also need to import the `Polly` namespace (`using Polly;`) to be able to use the `WaitAndRetryAsync` method.

## See Also

\* [Dynamic C# Client Proxies](Dynamic-CSharp-API-Clients.md)

## Swagger Integration

# Swagger Integration

[Swagger (OpenAPI)](https://swagger.io/) is a language-agnostic specification for describing REST APIs. It allows both computers and humans to understand the capabilities of a REST API without direct access to the source code. Its main goals are to:

- Minimize the amount of work needed to connect decoupled services.

- Reduce the amount of time needed to accurately document a service.

ABP Framework offers a prebuilt module for full Swagger integration with small configurations.

## Installation

> This package is already installed by default with the startup template. So, most of the time, you don't need to install it manually.

If installation is needed, it is suggested to use the [ABP CLI](../CLI.md) to install this package.

### Using the ABP CLI

Open a command line window in the folder of the `Web` or `HttpApi.Host` project (.csproj file) and type the following command:

```bash

abp add-package Volo.Abp.Swashbuckle

```

> If you haven't done it yet, you first need to install the [ABP CLI](../CLI.md). For other installation options, see [the package description page](https://abp.io/package-detail/Volo.Abp.Swashbuckle).

### Manual Installation

If you want to manually install;

1. Add the [Volo.Abp.Swashbuckle](https://www.nuget.org/packages/Volo.Abp.Swashbuckle) NuGet package to your `Web` or `HttpApi.Host` project:

`Install-Package Volo.Abp.Swashbuckle`

2. Add the `AbpSwashbuckleModule` to the dependency list of your module:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpSwashbuckleModule) // <-- Add module dependency like that

)]

public class YourModule : AbpModule

{

}

```

## Configuration

First, we need to use `AddAbpSwaggerGen` extension to configure Swagger in `ConfigureServices` method of our module:

```csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var services = context.Services;

//... other configurations.

services.AddAbpSwaggerGen(

options =>

{

options.SwaggerDoc("v1", new OpenApiInfo { Title = "Test API", Version = "v1" });

options.DocInclusionPredicate((docName, description) => true);

options.CustomSchemaIds(type => type.FullName);

}

);

}

```

Then we can use Swagger UI by calling `UseAbpSwaggerUI` method in the `OnApplicationInitialization` method of our module:

```csharp

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

//... other configurations.

app.UseAbpSwaggerUI(options =>

{

options.SwaggerEndpoint("/swagger/v1/swagger.json", "Test API");

});

//... other configurations.

}

```

### Hide ABP Endpoints on Swagger UI

If you want to hide ABP's default endpoints, call the `HideAbpEndpoints` method in your Swagger configuration as shown in the following example:

```csharp

services.AddAbpSwaggerGen(

options =>

{

//... other options

//Hides ABP Related endpoints on Swagger UI

options.HideAbpEndpoints();

}

)

```

## Using Swagger with OAUTH

For non MVC/Tiered applications, we need to configure Swagger with OAUTH to handle authorization.

> ABP Framework uses OpenIddict by default. To get more information about OpenIddict, check this [documentation](../Modules/OpenIddict.md).

To do that, we need to use `AddAbpSwaggerGenWithOAuth` extension to configure Swagger with OAuth issuer and scopes in `ConfigureServices` method of our module:

```csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var services = contex.Services;

//... other configarations.

services.AddAbpSwaggerGenWithOAuth(

"https://localhost:44341", // authority issuer

new Dictionary<string, string> //

{ // scopes

{"Test", "Test API"} //

}, //

options =>

{

options.SwaggerDoc("v1", new OpenApiInfo { Title = "Test API", Version = "v1" });

options.DocInclusionPredicate((docName, description) => true);

options.CustomSchemaIds(type => type.FullName);

}

);

}

```

Then we can use Swagger UI by calling `UseAbpSwaggerUI` method in the `OnApplicationInitialization` method of our module:

```csharp

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

//... other configurations.

app.UseAbpSwaggerUI(options =>

{

options.SwaggerEndpoint("/swagger/v1/swagger.json", "Test API");

var configuration = context.ServiceProvider.GetRequiredService<IConfiguration>();

options.OAuthClientId("Test\_Swagger"); // clientId

options.OAuthClientSecret("1q2w3e\*"); // clientSecret

});

//... other configurations.

}

```

> Do not forget to set `OAuthClientId` and `OAuthClientSecret`.

## Using Swagger with OIDC

You may also want to configure swagger using **\*\*OpenIdConnect\*\*** instead of OAUTH. This is especially useful when you need to configure different metadata address than the issuer in cases such as when you deploy your application to kubernetes cluster or docker. In these cases, metadata address will be used in sign-in process to reach the valid authentication server discovery endpoint over the internet and use the internal network to validate the obtained token.

To do that, we need to use `AddAbpSwaggerGenWithOidc` extension to configure Swagger with OAuth issuer and scopes in `ConfigureServices` method of our module:

```csharp

context.Services.AddAbpSwaggerGenWithOidc(

configuration["AuthServer:Authority"],

scopes: new[] { "SwaggerDemo" },

// "authorization\_code"

flows: new[] { AbpSwaggerOidcFlows.AuthorizationCode },

// When deployed on K8s, should be metadata URL of the reachable DNS over internet like https://myauthserver.company.com

discoveryEndpoint: configuration["AuthServer:Authority"],

options =>

{

options.SwaggerDoc("v1", new OpenApiInfo { Title = "SwaggerDemo API", Version = "v1" });

options.DocInclusionPredicate((docName, description) => true);

options.CustomSchemaIds(type => type.FullName);

});

```

The `flows` is a list of default oidc flows that is supported by the oidc-provider (authserver). You can see the default supported flows below:

- `AbpSwaggerOidcFlows.AuthorizationCode`: The `"authorization\_code"` flow is the **\*\*default and suggested\*\*** flow. **\*\*Doesn't require a client secret\*\*** when even there is a field for it.

- `AbpSwaggerOidcFlows.Implicit`: The deprecated `"implicit"` flow that was used for javascript applications.

- `AbpSwaggerOidcFlows.Password`: The legacy `password` flow which is also known as Resource Ownder Password flow. You need to provide a user name, password and client secret for it.

- `AbpSwaggerOidcFlows.ClientCredentials`: The `"client\_credentials"` flow that is used for server to server interactions.

You can define one or many flows which will be shown in the Authorize modal. You can set it **\*\*null which will use the default "authorization\_code"\*\*** flow.

The `discoveryEndpoint` is the reachable openid-provider endpoint for the `.well-known/openid-configuration`. You can set it to **\*\*null which will use default AuthServer:Authority\*\*** appsettings configuration. If you are deploying your applications to a kubernetes cluster or docker swarm, you should to set the `discoveryEndpoint` as real DNS that should be reachable over the internet.

> If are having problems with seeing the authorization modal, check the browser console logs and make sure you have a correct and reachable `discoveryEndpoint`

# User Interface

## MVC / Razor Pages

### Overall

### Navigation / Menus

### Forms & Validation

### Modals

### Data Tables

### Auto-Complete Select

### Page Alerts

### Dynamic JavaScript API Client Proxies

### Static JavaScript API Client Proxies

### Client Side Package Management

### Bundling & Minification

### Tag Helpers

#### Form Elements

#### Dynamic Forms

### Widgets

### Toolbars

### Page Header

### Branding

### Layout Hooks

### Testing

### Theming

#### The Basic Theme

#### LeptonX Lite

### JavaScript API

#### Overall

#### Localization

#### Auth

#### Current User

#### Settings

#### Features

#### AJAX

#### Message

#### Notify

#### Block/Busy

#### Events

#### DOM

#### Logging

#### Resource Loader

### Customize/Extend the UI

#### Overall

#### Entity Action Extensions

#### Data Table Column Extensions

#### Page Toolbar Extensions

### Security

#### Security Headers

## Blazor

### Overall

### Navigation / Menu

### Localization

### Theming

#### Overall

#### The Basic Theme

#### LeptonX Lite

#### Branding

#### Page Header

#### Page Layout

#### Toolbars

### Security

#### Authentication

#### Authorization

### Services

#### Current User

#### Current Tenant

#### Notification

#### Message

#### Page Alerts

#### Page Progress

### Other Components

#### SubmitButton

### Settings

### Error Handling

### Customization / Overriding Components

### Global Scripts & Styles

### Global Features

### Routing

### PWA Configuration

### Layout Hooks

## Angular

### Quick Start

### Development

#### Environment Variables

#### About Feature Libraries

#### Service Proxies

#### PWA Configuration

#### Unit Testing

### Core Functionality

#### Config State Service

#### Authorization

#### Current User

#### HTTP Requests

#### Localization

#### Form Validation

#### Settings

#### Features

#### Global Features

#### Permission Management

#### Multi Tenancy

#### Account Module

### Utilities

#### Managing RxJS Subscriptions

#### Working with Lists

#### Easy \*ngFor trackBy

#### Router Events

#### UI/Angular/Router-Events.md

#### Inserting Scripts & Styles to DOM

#### Lazy Loading Scripts & Styles

#### Projecting Angular Content

#### Modal

#### Confirmation Popup

#### Loading Directive

#### Toast Overlay

#### Page Alerts

#### Ellipsis

#### Context Strategy

#### Cross Origin Strategy

#### Dom Strategy

#### Container Strategy

#### Content Security Strategy

### Customization

#### Customization Guide

#### Theming

##### Overall

##### Configuration

##### The Basic Theme

##### LeptonX Lite

#### Modifying the Menu

#### Component Replacement

#### Extensions

##### Overall

##### Entity Action Extensions

##### Data Table Column Extensions

##### Page Toolbar Extensions

##### Dynamic Form Extensions

##### Date, time and datetime format pipes

### Components

#### Page

#### Chart

#### Card

## React Native

### Getting Started

## Common

#### Themes

#### Overriding the User Interface

#### Utilities

##### Linked List (Doubly)

# Data Access

## Overall

## Entity Framework Core

### Database Migrations

### Switch DBMS

#### To MySQL

#### To PostgreSQL

#### To Oracle

##### Oracle (Official)

##### Oracle (Devart)

#### To SQLite

## MongoDB

## Dapper

# Real Time

## SignalR Integration

# Dapr Integration

# Testing

# Deployment

## Configuring OpenIddict

## Configuring for Production

## Deploying to a Clustered Environment

## Distributed / Microservice Solutions

## Optimizing for Production

# Application Modules

## Overall

## Account

## Audit Logging

## Background Jobs

## CMS Kit

## Docs

## Feature Management

## Identity

## IdentityServer

### IdentityServer Migration Guide

## OpenIddict

### OpenIddict Migration Guide

## Permission Management

## Setting Management

## Tenant Management

## Virtual File Explorer

## Common

### Database Tables

# Samples

## All Samples

## eShopOnAbp

## EventHub

## Microservice Demo (legacy)

# Books

## Mastering ABP Framework

## Implementing Domain Driven Design

# Release Information

## Upgrading

## Official Packages

## Preview Releases

## Nightly Builds

## Road Map

## Migration Guides

# Contribution Guide