abp.io

Building Microservice Solutions

with the ABP Framework

By

Halil İbrahim Kalkan

Galip Tolga Erdem

Enis Necipoğlu

About the Authors

Halil İbrahim Kalkan is a computer engineer who loves building reusable libraries, creating distributed solutions, and working on software architectures. He is an expert in domain-driven design, multi-tenancy, modularity, and microservice architecture. Halil has been building software since 1997 (when he was 14) and working as a professional since 2007. He has a lot of articles and talks on software development. He is a very active open source contributor and has built many projects based on web and Microsoft technologies. Halil is currently leading the open source ABP Framework, which provides a complete architectural solution for building .NET applications.

Galip Tolga Erdem is a computer engineer who loves creating distributed systems and working on software architectures. He excels in software authentication and authorization, domain-driven design, modularity, microservice architecture, and DevOps. Galip Tolga has been building software for enterprise banking, medical and high education systems. He is working as a professional since 2011. He doesn't like to be active on social platforms and mostly contributes to private repositories. He has built many enterprise projects with different technologies but he loves Microsoft technologies after .NET Core. Galip Tolga is currently working as a senior software engineer with Halil İbrahim.

Enis Necipoğlu is a software developer who loves contributing and maintaining open-source projects. He is a self-learner developer and started programming at age 14. He follows Microsoft technologies closely. He is experienced on migrating monolith to micro-service and worked on projects that under heavy request traffic. Enis worked on do it yourself app making platform and prefers working on projects from developer to developer. He currently works on ABP Framework alongside Halil İbrahim Kalkan & Galip Tolga Erdem.

Table of Contents

Technical Requirements

The eShopOnAbp contains an angular application, .Net Razor public application, various microservices, and gateways. While some of the microservices use RDBMS, some of them use no-SQL and caching databases. As in all the distributed systems, this variety of tech-stack has some pre-requirements.

For development and running the solution:

- NPM v14+ (or Yarn 1.20+) for the angular application
- .NET 6.0 SDK for .NET applications and microservices
- Powershell 5.0+ for running scripts
- Docker for Desktop v3.0+ for running the databases and external third-party services like RabbitMq
- Dotnet Tye for running the eShopOnAbp solution
- Visual Studio 2022 or another suitable IDE that supports .NET6.0

For running the solution on the local Kubernetes cluster:

- Docker for Desktop with Kubernetes support or Minikube

For running the solution on Docker:

- Docker for Desktop with docker-compose support

Chapter 1: Introduction

This chapter introduces the eShopOnAbp project and explains both the overall structure and the main components of the solution.

This chapter consists of the following topics:

- Introducing the eShopOnAbp Solution
- Comparing with Other Examples
- Understanding the Big Picture
- Exploring the Solution and the Folder Structure

Introducing the eShopOnAbp Solution

I suppose that everyone will accept that microservice architecture is challenging by its nature. It promises strong boundaries between services, allows you to develop, deploy and scale services independently, and uses the proper technology for each service. However, dealing with communication and integration problems of distributed components, the operational complexity of deploying and monitoring many services and applications are some of the fundamental costs that come with microservice architecture.

There are a lot of books and articles written, and many talks performed by the experts in that area. However, hand-on design and development of a microservice solution requires us to make many decisions about the architecture and implementation of the system and solve many problems already thought or solved by someone else before. At that point, a concrete example solution has a huge value. While every microservice solution will have different requirements and restrictions that effect the implementation somehow, there are still a lot of lessons that can be learned from a complete, concrete and documentation-referenced example solution.

eShopOnAbp is a reference microservice solution that demonstrates how you can build a microservice solution using the ABP Framework and ASP.NET Core.

Comparing with Other Examples

There are already some microservice solutions in the .NET ecosystem. eShopOnContainers is one of the most famous ones. It's been created by Microsoft. It is well documented and shows many different architectural patterns and use cases in a single solution. If you haven't check it yet, you should explore its source code and documentation. They have also created a few e-books based on or related to this project.

One problem with the eShopOnContainers solution is that it is a draft solution and not so production ready. Microsoft tried to not depend on specific libraries or frameworks for implementing patterns like DDD, CQRS, event sourcing, event bus, etc. As a result of that, the majority of the code base consists of simplified infrastructural implementations of the architectures, which makes the application code over-complicated and the business code is mixed with the infrastructural code. It is a reasonable decision, but makes it less reusable and less usable for real-life solutions.

Microsoft has built another reference solution, which is this time based on its Dapr project: eShopOnDapr. This is essentially a simplified re-implementation of the eShopOnContainers using Dapr to implement distributed application development patterns, like service discovery, synchronous and asynchronous communication of services. This project can be considered more production ready since most of the built-in infrastructural code in the eShopOnContainers project has been replaced by the Dapr library, which results the codebase to be cleaner and more maintainable.

eShopOnAbp is inspired by these two projects and takes them one step further as being a production ready solution with minimal or no infrastructural code in the solution. Since the ABP Framework is a full-stack and microservice-compatible open source web application framework, it offers built-in solutions to common microservice architecture patterns and also provides infrastructure for common business application development requirements. As a result, your codebase becomes clean, understandable and maintainable.

Understanding the Big Picture

The eShopOnAbp solution consists of several services, applications, API gateways, databases and some other components as shown in the following figure:

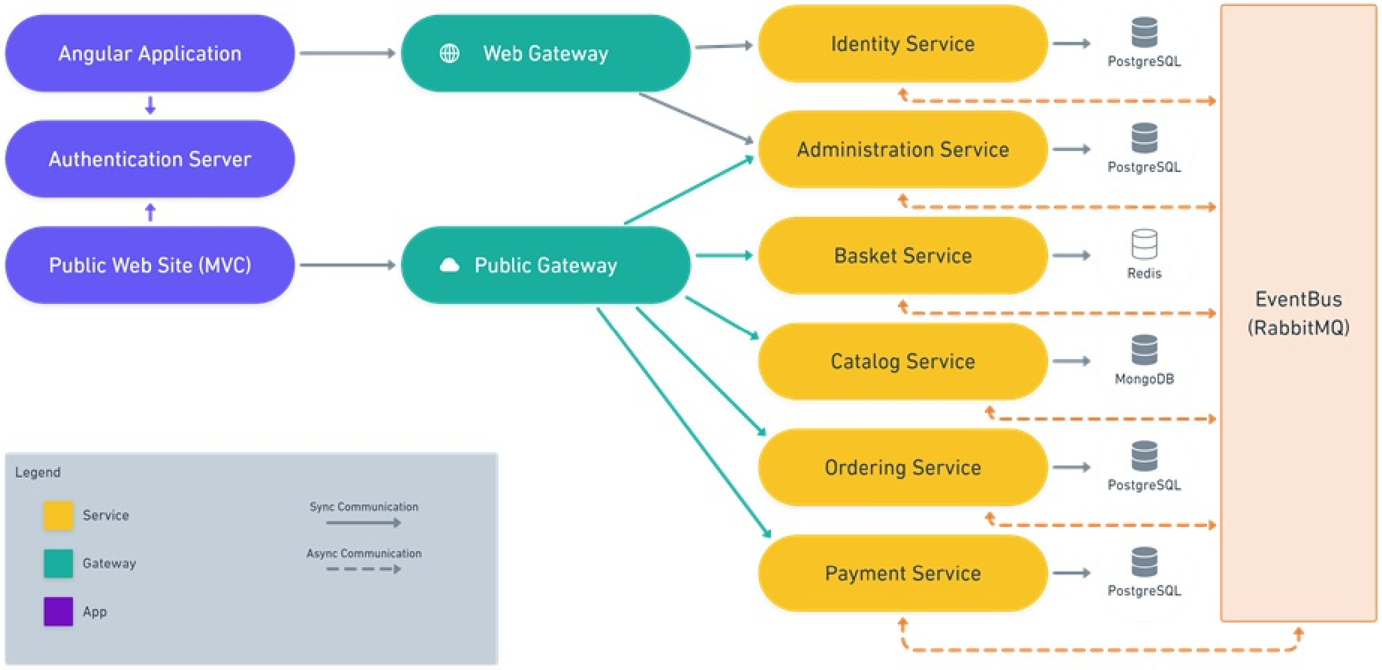
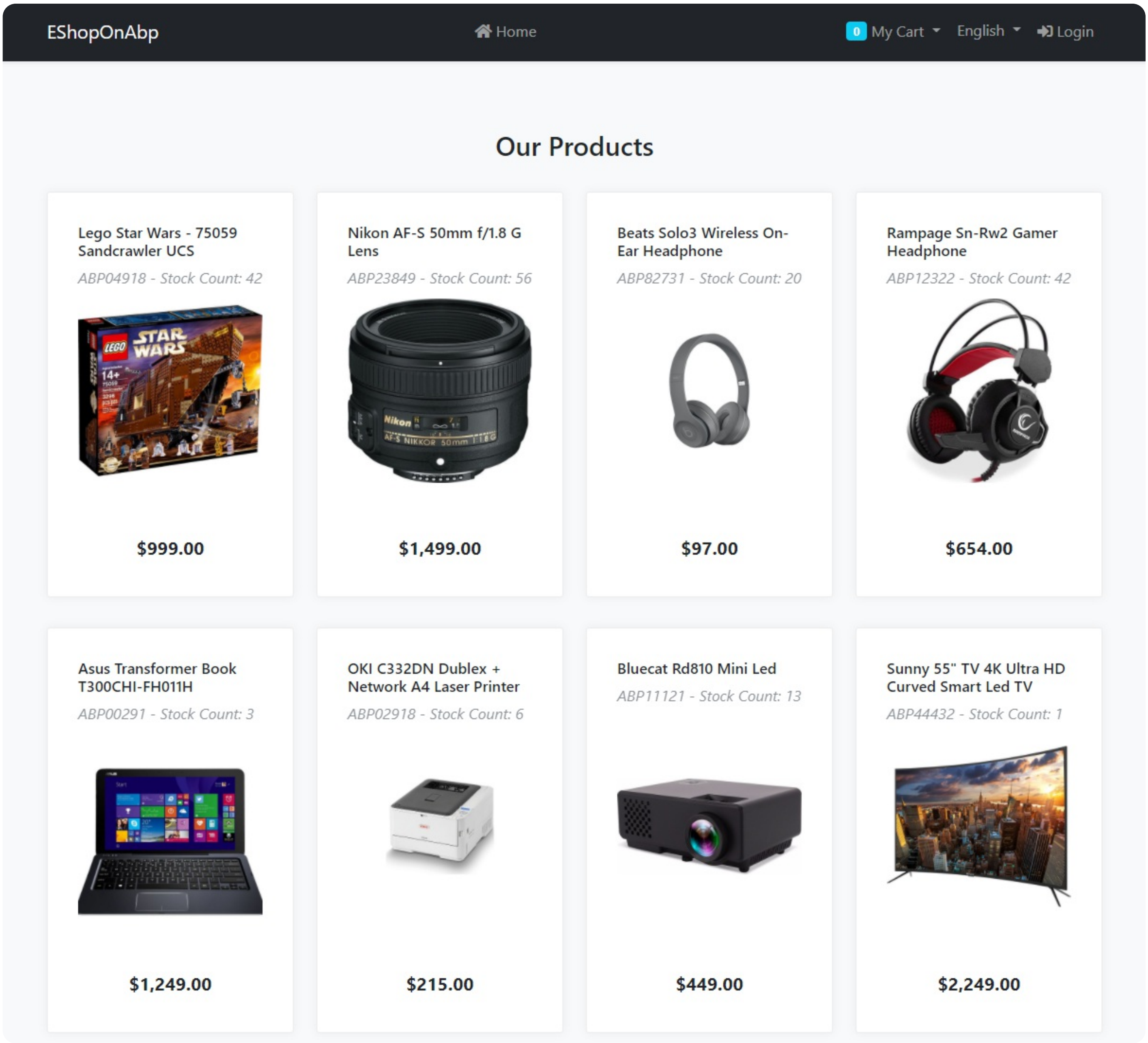


Figure 1.1: eShopOnAbp overall

The following subsections explains these components in brief.

The Public Website

The public website is the essential web application that is used by the end-users to browse products, add them to basket, place the order and monitor the order status. Here's a screenshot from the product page



Our Products

Lego Star Wars - 75059 Sandcrawler UCS
ABP04918 - Stock Count: 42
$999.00

Nikon AF-S 50mm f/1.8 G Lens
ABP23849 - Stock Count: 56
$1,499.00

Beats Solo3 Wireless On-Ear Headphone
ABP82731 - Stock Count: 20
$97.00

Rampage Sn-Rw2 Gamer Headphone
ABP12322 - Stock Count: 42
$654.00

Asus Transformer Book T300CHI-FH011H
ABP00291 - Stock Count: 3
$1,249.00

OKI C332DN Dublex + Network A4 Laser Printer
ABP02918 - Stock Count: 6
$215.00

Bluecat Rd810 Mini Led
ABP11121 - Stock Count: 13
$449.00

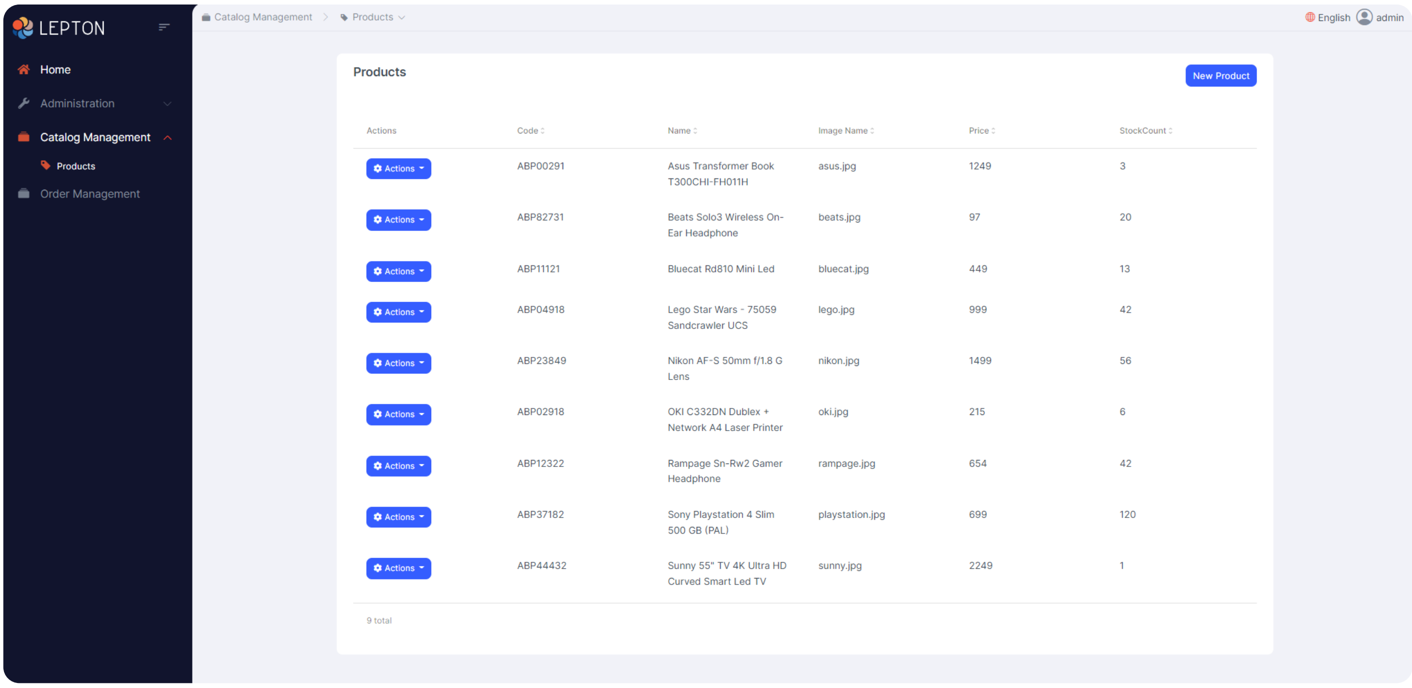
Sunny 55" TV 4K Ultra HD Curved Smart Led TV
ABP44432 - Stock Count: 1
$2,249.00

Figure 1.2: eShopOnAbp public web application products list page

We will explore this application in *Chapter #4: Exploring the Applications*.

The Angular (admin) Application

The angular application is used by the admin users to manage products, orders, users, permissions and other details. Here's a screenshot from the product management page:



Products

New Product

Actions	Code	Name	Image Name	Price	StockCount
Actions	ABP00291	Asus Transformer Book T300CHI-FH011H	asus.jpg	1249	3
Actions	ABP82731	Beats Solo3 Wireless On-Ear Headphone	beats.jpg	97	20
Actions	ABP11121	Bluecat Rd810 Mini Led	bluecat.jpg	449	13
Actions	ABP04918	Lego Star Wars - 75059 Sandcrawler UCS	lego.jpg	999	42
Actions	ABP23849	Nikon AF-S 50mm f/1.8 G Lens	nikon.jpg	1499	56
Actions	ABP02918	OKI C332DN Dublex + Network A4 Laser Printer	oki.jpg	215	6
Actions	ABP12322	Rampage Sn-Rw2 Gamer Headphone	rampage.jpg	654	42
Actions	ABP37182	Sony Playstation 4 Slim 500 GB (PAL)	playstation.jpg	699	120
Actions	ABP44432	Sunny 55" TV 4K Ultra HD Curved Smart Led TV	sunny.jpg	2249	1

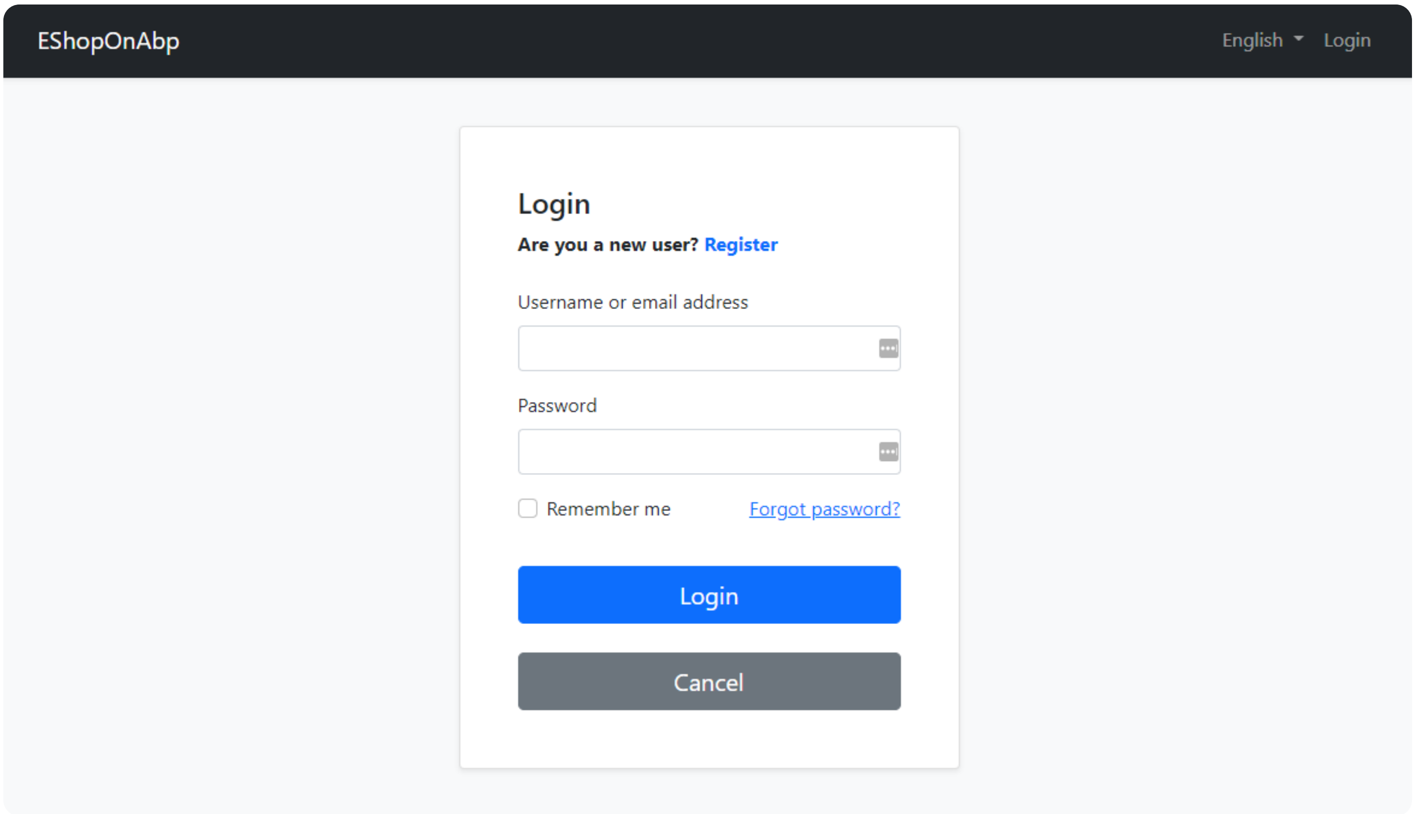
9 total

Figure 1.3: eShopOnAbp admin application products management page

We will explore this application in *Chapter #4: Exploring the Applications*.

The Authentication Server Application

The authentication server application that provides single-sign-on functionality and authentication token endpoint for other applications and services. The two web applications redirect the user to the authentication server on login. Here's a screenshot from the application login page:



Login

Are you a new user? Register

Username or email address

Password

□ Remember me

Forgot password?

Figure 1.4: eShopOnAbp authentication server

We will explore the authentication system in *Chapter #3: Understanding the Authentication System*.

The API Gateways

An API Gateway is used to provide a single entry point to the system for the client applications. eShopOnAbp implements the BFF (Backend for Frontend) pattern, where you provide a separate API gateway for each client application. eShopOnAbp has the following API gateways:

- **Public Web Gateway**: This is for the API gateway of the Public Website application, where end-users are using the system.
- **Web Gateway**: This API gateway is used by the Angular (admin) application.

These API gateways will be explained in *Chapter #5: Understanding the API Gateways*.

The Services

The solution contains the following microservices:

- **Catalog service**: Stores all the products information and their stock counts and performs the business logic based on these data. Uses a MongoDB database.
- **Basket service**: Used to keep track of the baskets of the users. It stores the state in Redis as a temporary data store. Basket data of users are lost if they don't access it for a while (a week by default). Performs synchronous gRPC API calls to the Catalog service and listens for events from that service for product updates.
- **Ordering service**: Used to place and keep track of an order. An order is typically created by a user by submitting the basket items. This service uses a separate PostgreSQL database.
- **Payment service**: This service is integrated to Paypal, executes the payment requests and listens for payment events from Paypal. This service uses a separate PostgreSQL database.
- **Administration service**: This service is mainly a wrapper for ABP's pre-built Permission Management, Setting Management and Audit Logging application modules. These are shared and cross-cutting functionalities available to all microservices and used to manage the system. This service uses a separate PostgreSQL database.
- **Identity service**: This service is mainly a wrapper for ABP's pre-built Identity and IdentityServer application modules. It is used to manage users, roles and clients in the system. This service uses a separate PostgreSQL database that is shared by the *Authentication Server* application.

We will explain these services with details in *Chapter #6: Developing the Microservices*.

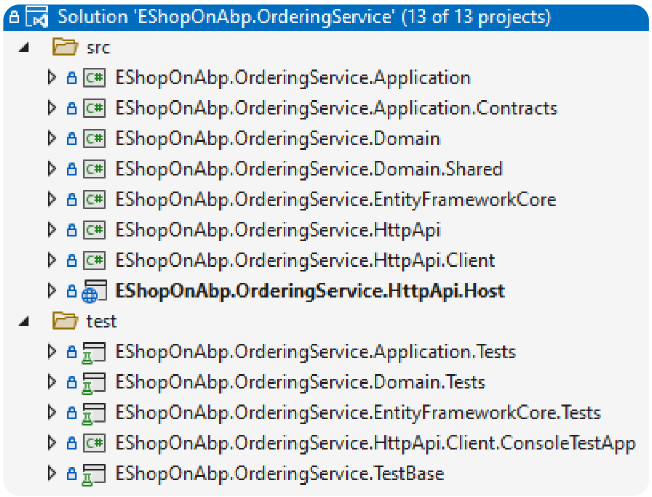
Exploring the Solution and the Folder Structure

Once you clone the eShopOnAbp repository (will be covered in *Chapter #2: Running the solution*), you will see the following folders in the root directory:

- apps: Contains the web applications:
  - angular: The Angular (admin) web application.

  - public-web: The public website that is used by the end-users of the system, where they make shopping. It has its own .NET solution (.sln).
  - auth-server: The authentication server application that provides a single-sign-on functionality and authentication token endpoint for other applications and services. The two web applications redirect the user to the authentication server on login. It has its own .NET solution (.sln).
- gateways: Contains the API gateway solutions.
  - web: Contains the .NET solution (.sln) for the API gateway that is used by the Angular (admin) application.
  - public-web: Contains the .NET solution (.sln) for the API gateway that is used by the public website.
- services: Contains the microservice solutions. Every microservice has its own .NET solution (.sln), so they can be independently developed.
  - administration: .NET solution (.sln) for the Administration microservice.
  - basket: .NET solution (.sln) for the Basket microservice.
  - catalog: .NET solution (.sln) for the Catalog microservice.
  - identity: .NET solution (.sln) for the Identity microservice.
  - ordering: .NET solution (.sln) for the Ordering microservice.
  - payment: .NET solution (.sln) for the Payment microservice.
- shared: Contains some shared projects to collect the common code across services, API gateways and other applications.
- ect: Contains scripts, configuration and some other files to build, deploy and run the solution.

As explained in the list above, there are many .NET solutions in eShopOnAbp. In this way, every solution can be separately developed, deployed and tested. For example, the following figure shows the Ordering microservice in Visual Studio:

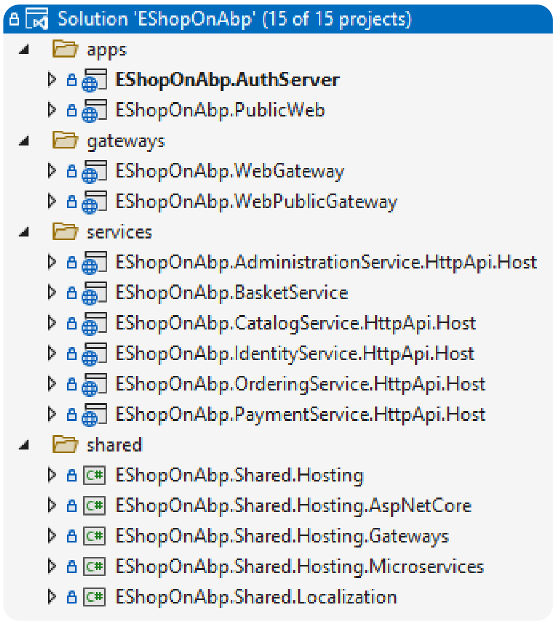


Solution 'EShopOnAbp.OrderingService' (13 of 13 projects)
- src
  - EShopOnAbp.OrderingService.Application
  - EShopOnAbp.OrderingService.Application.Contracts
  - EShopOnAbp.OrderingService.Domain
  - EShopOnAbp.OrderingService.Domain.Shared
  - EShopOnAbp.OrderingService.EntityFrameworkCore
  - EShopOnAbp.OrderingService.HttpApi
  - EShopOnAbp.OrderingService.HttpApi.Client
  - **EShopOnAbp.OrderingService.HttpApi.Host**
- test
  - EShopOnAbp.OrderingService.Application.Tests
  - EShopOnAbp.OrderingService.Domain.Tests
  - EShopOnAbp.OrderingService.EntityFrameworkCore.Tests
  - EShopOnAbp.OrderingService.HttpApi.Client.ConsoleTestApp
  - EShopOnAbp.OrderingService.TestBase

Figure 1.2: .NET solution of the Ordering microservice

The Ordering service is a layered solution, and also contains test projects for each layer. The EShopOnAbp.OrderingService.HttpApi.Host project is the essential project that runs the microservice.

There is also a **main solution** that contains the executable projects (.csproj) of each solution. Using the main solution, you can open, run and debug all of the executable applications (microservices, API gateways and web applications) in a single place. The main solution's name is EShopOnAbp.sln and it's located in the root folder of the solution. The following figure shows the main solution in Visual Studio:



```
Solution 'EShopOnAbp' (15 of 15 projects)
  apps
    EShopOnAbp.AuthServer
    EShopOnAbp.PublicWeb
  gateways
    EShopOnAbp.WebGateway
    EShopOnAbp.WebPublicGateway
  services
    EShopOnAbp.AdministrationService.HttpApi.Host
    EShopOnAbp.BasketService
    EShopOnAbp.CatalogService.HttpApi.Host
    EShopOnAbp.IdentityService.HttpApi.Host
    EShopOnAbp.OrderingService.HttpApi.Host
    EShopOnAbp.PaymentService.HttpApi.Host
  shared
    EShopOnAbp.Shared.Hosting
    EShopOnAbp.Shared.Hosting.AspNetCore
    EShopOnAbp.Shared.Hosting.Gateways
    EShopOnAbp.Shared.Hosting.Microservices
    EShopOnAbp.Shared.Localization
```

Figure 1.3: The main .NET solution

Chapter #2: Running the Solution will explain how to run all the applications using this solution.

Summary

In this chapter, you've learned the main components of the eShopOnAbp solution and their relations to each other, in brief. They all will be deeply explained in the next chapters.

As we've understood the solution overall, it is time to run the solution and explore the applications and services as explained in the next chapter.

Chapter 2: Running the solution

This chapter explains the various ways to run the solution. It also discusses the ways of running third party applications. By the end of the chapter, you will learn how to run the solution using Microsoft's Tye project in your local environment and using Helm charts to run the solution on Docker Desktop as a staging environment.

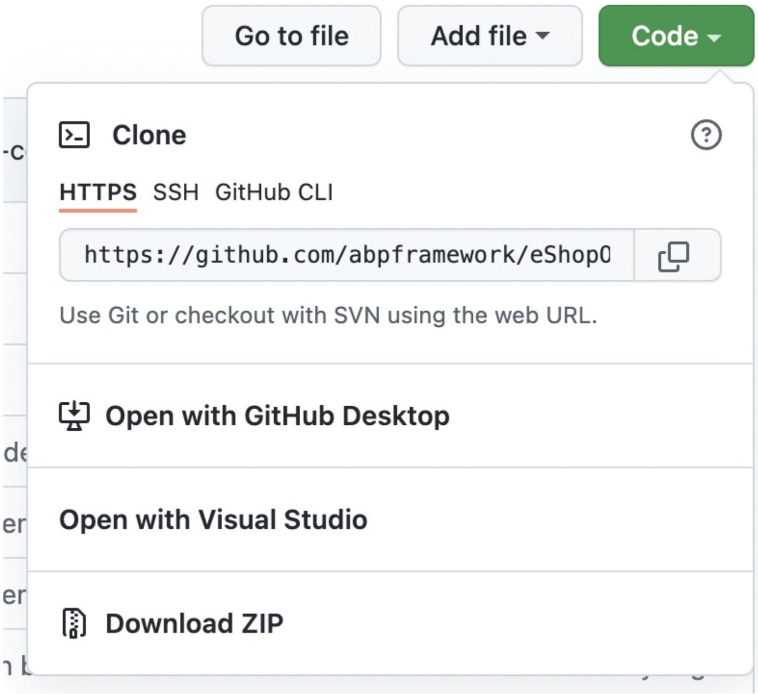
This chapter consists of the following topics:

- Cloning the eShopOnAbp repository
- Setting up the environment
- Using Visual Studio (F5 experience) to run the solution
- Using the Tye project to run the solution
- Using a local Kubernetes cluster to run the solution

Cloning the eShopOnAbp repository

A git repository can be cloned in a couple of different ways. The easiest one is cloning with Git CLI. Some softwares such as GitHub Desktop, VS Code and most of IDEs provide a GUI to perform git operations.

Navigate to eShopOnAbp Repository on GitHub. There is a **"Code"** button at the top right corner that shows some cloning ways:



https://github.com/abpframework/eShopO

Use Git or checkout with SVN using the web URL.

Figure 2.1: GitHub Code button menu

The project can be cloned directly using Open with GitHub Desktop or Open with Visual Studio buttons.

If any of these applications are not installed, you can continue with command-line commands. HTTPS and SSH addresses can be used for cloning with Git CLI. Before starting, Git CLI needs to be installed properly.

The following commands clone the repository to your local computer:

```
git clone https://github.com/abpframework/eShopOnAbp.git

git clone git@github.com:abpframework/eShopOnAbp.git
```

> Read more about cloning a repository from here

Setting up the environment

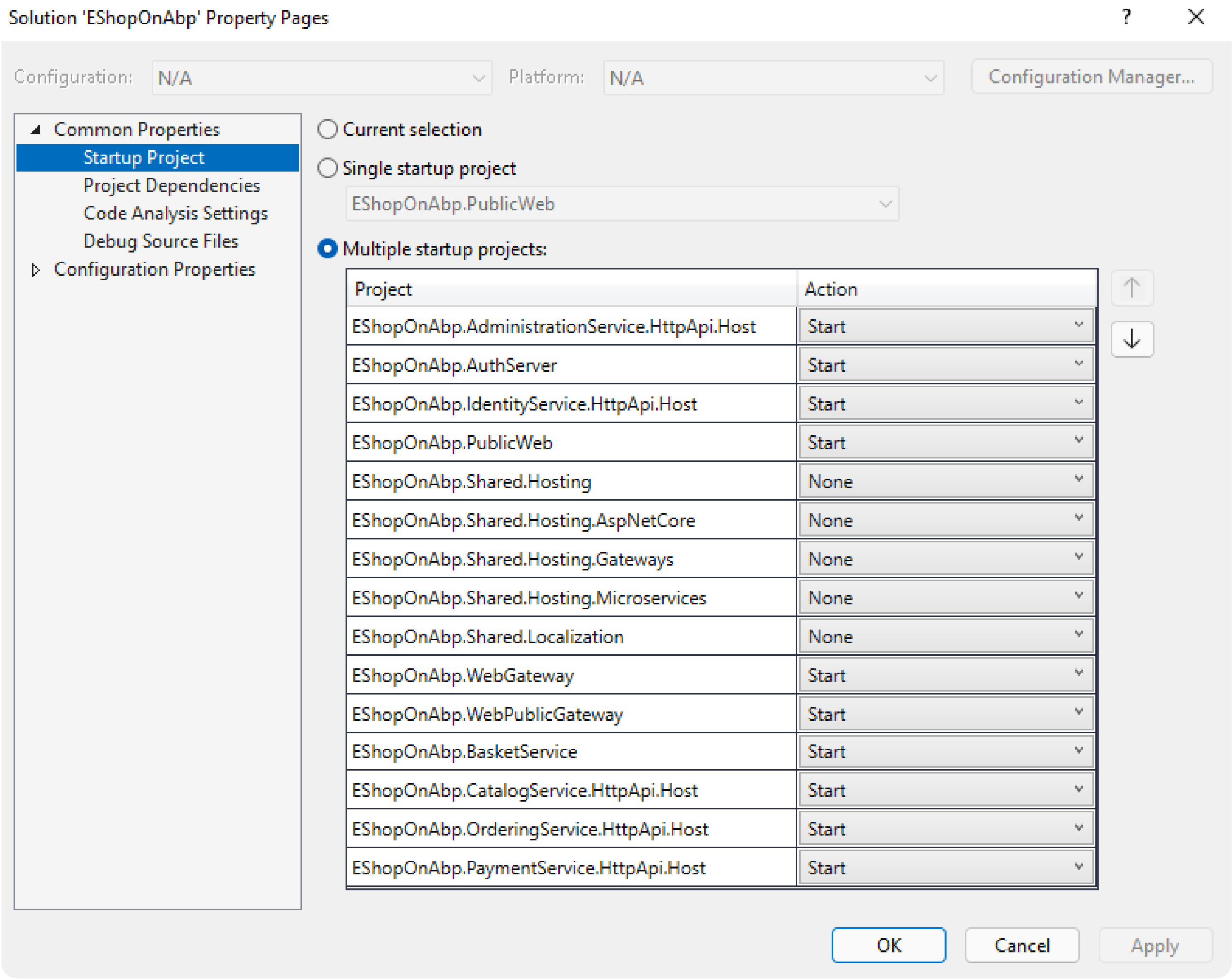
EShopOnAbp is built with ABP Framework on .NET and Angular. Before running the solution, tech stack of eShopOnAbp must be installed. Make sure following technologies are installed properly:

- .NET SDK
- Nodejs and yarn

Using Visual Studio (F5 experience) to run the solution

This section will explain running the solution in a classical way that is Visual Studio.

1. You should make the development environment running first by executing etc/docker/up.ps1.
2. Open the solution file named EShopOnAbp.sln with Visual Studio.
3. Right click the EShopOnAbp solution and click the **Properties**
4. Choose Multiple startup projects radio button and make all the actions Start for the executable projects only. *(all of them except shared folder)*



Solution 'EShopOnAbp' Property Pages

Configuration: N/A Platform: N/A

○ Current selection

○ Single startup project

EShopOnAbp.PublicWeb

● Multiple startup projects:

Project	Action
EShopOnAbp.AdministrationService.HttpApi.Host	Start
EShopOnAbp.AuthServer	Start
EShopOnAbp.IdentityService.HttpApi.Host	Start
EShopOnAbp.PublicWeb	Start
EShopOnAbp.Shared.Hosting	None
EShopOnAbp.Shared.Hosting.AspNetCore	None
EShopOnAbp.Shared.Hosting.Gateways	None
EShopOnAbp.Shared.Hosting.Microservices	None
EShopOnAbp.Shared.Localization	None
EShopOnAbp.WebGateway	Start
EShopOnAbp.WebPublicGateway	Start
EShopOnAbp.BasketService	Start
EShopOnAbp.CatalogService.HttpApi.Host	Start
EShopOnAbp.OrderingService.HttpApi.Host	Start
EShopOnAbp.PaymentService.HttpApi.Host	Start

Figure 2.2: GitHub Code button menu

5. Run the solution by hitting the run button *(▶□)*.

6. Wait until all the migrations are done. *(approximately 30 seconds for the first time.)*

7. Then you can navigate to public web app. (localhost:44335)

Using the Tye project to run the solution

This section is about running the solution with Tye. Project Tye is a developer tool that makes developing, testing, and deploying micro-services and distributed applications easier.

Installing the tools

1. Project Tye should be installed by following these steps.

2. Powershell has to be installed if it wasn't. You can find powershell releases by visiting the PowerShell/PowerShell repository on GitHub. The installation documentation can also be found from here. Powershell installation and version can be checked with the following command:

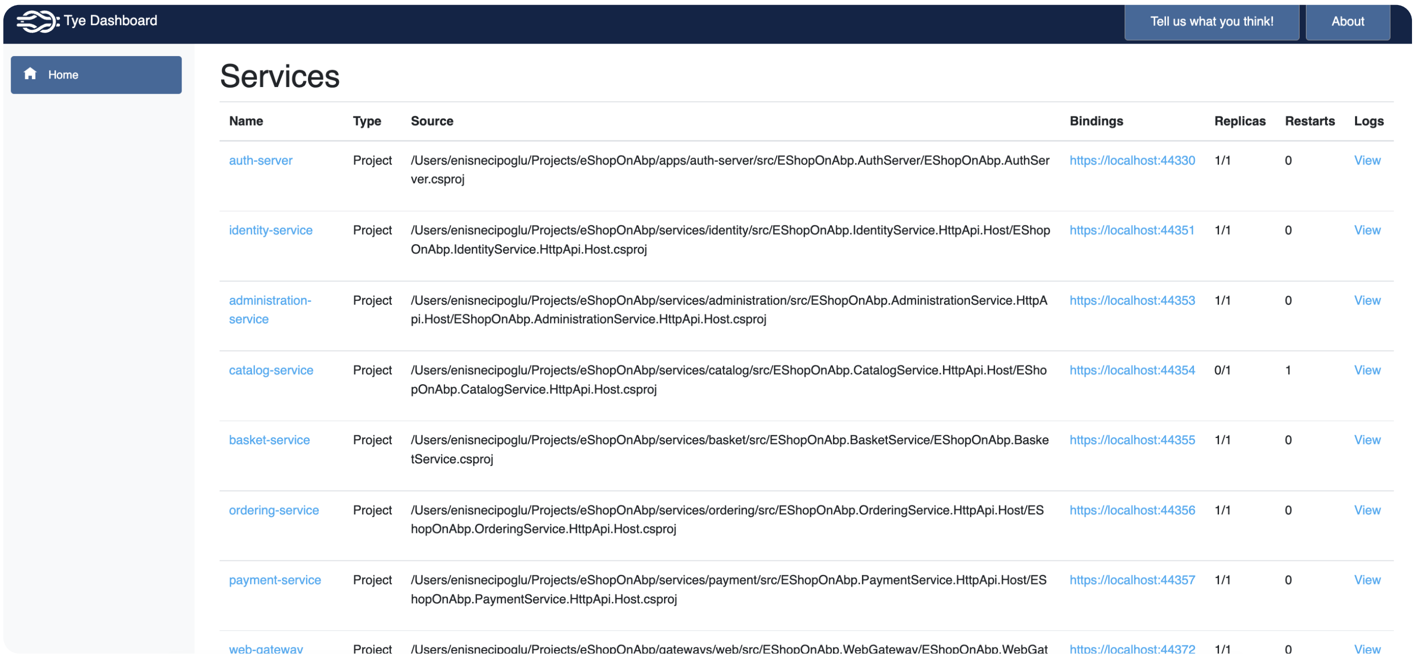
```
pwsh -v
```

PowerShell is pre-installed on Windows operation system. If you're on a Windows environment, you can skip the step 2.

Running the solution

The solution includes 11 runnable projects and the application environment requires some applications such as MongoDB and PostgreSQL databases, Redis, RabbitMQ instances. Running all of them one by one takes a lot of time and it's not sustainable. So, the repository has a file named run-tye.ps1 to run the entire environment and applications.

1. Run the run-tye.ps1
2. Navigate to localhost:8000 and check running applications.
3. Wait until each application is running (1/1)



Services

Name	Type	Source	Bindings	Replicas	Restarts	Logs
auth-server	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/apps/auth-server/src/EShopOnAbp.AuthServer/EShopOnAbp.AuthServer.csproj	https://localhost:44330	1/1	0	View
identity-service	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/services/identity/src/EShopOnAbp.IdentityService.HttpApi.Host/EShopOnAbp.IdentityService.HttpApi.Host.csproj	https://localhost:44351	1/1	0	View
administration-service	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/services/administration/src/EShopOnAbp.AdministrationService.HttpApi.Host/EShopOnAbp.AdministrationService.HttpApi.Host.csproj	https://localhost:44353	1/1	0	View
catalog-service	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/services/catalog/src/EShopOnAbp.CatalogService.HttpApi.Host/EShopOnAbp.CatalogService.HttpApi.Host.csproj	https://localhost:44354	0/1	1	View
basket-service	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/services/basket/src/EShopOnAbp.BasketService/EShopOnAbp.BasketService.csproj	https://localhost:44355	1/1	0	View
ordering-service	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/services/ordering/src/EShopOnAbp.OrderingService.HttpApi.Host/EShopOnAbp.OrderingService.HttpApi.Host.csproj	https://localhost:44356	1/1	0	View
payment-service	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/services/payment/src/EShopOnAbp.PaymentService.HttpApi.Host/EShopOnAbp.PaymentService.HttpApi.Host.csproj	https://localhost:44357	1/1	0	View
web-gateway	Project	/Users/enisnecipoglu/Projects/eShopOnAbp/gateways/web/src/EShopOnAbp.WebGateway/EShopOnAbp.WebGat	https://localhost:44372	1/1	0	View

Figure 2.3: Tye dashboard preview

It may take some time running the first time.

Using a local Kubernetes cluster to run the solution

This section explains running the solution in a local Kubernetes cluster with using Kubernetes support on docker desktop using helm.

Installing the tools

Follow the steps below to install the necessary tools to your local computer:

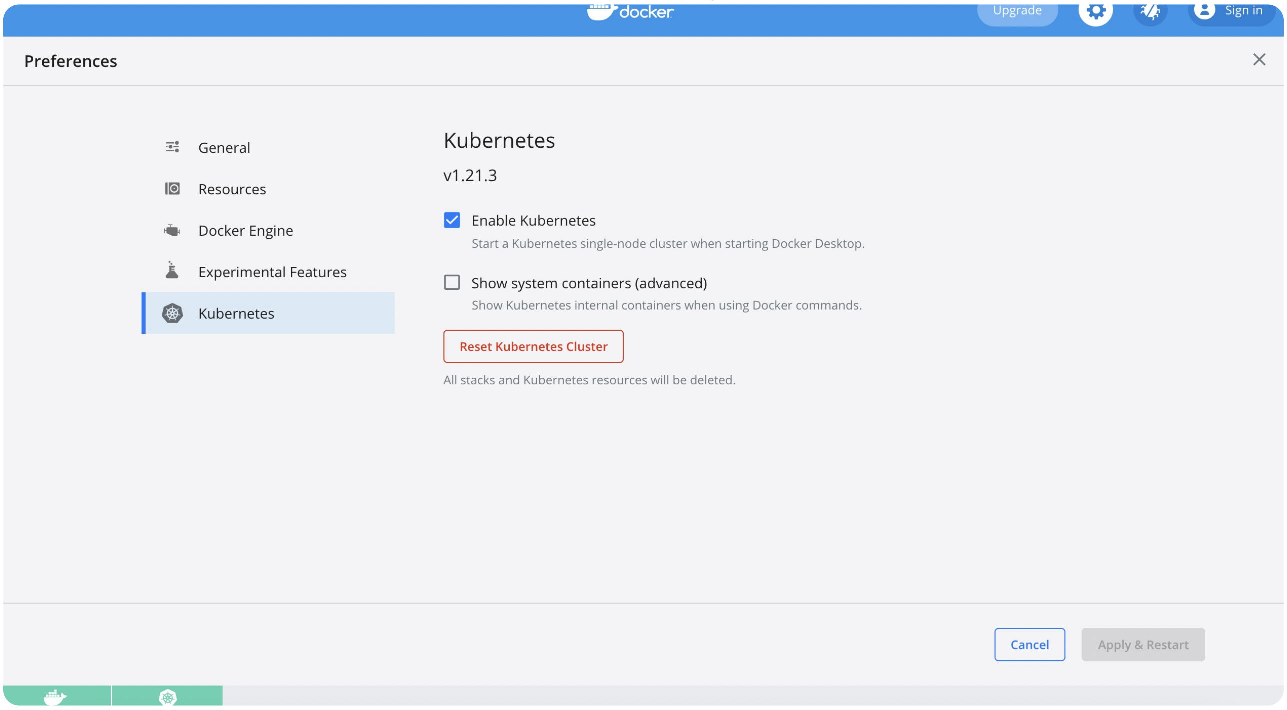
1. Install Docker Desktop if not installed before

2. Install Powershell if not installed before

3. Enable Kubernetes support on Docker Desktop

   - Launch the Docker Desktop app and navigate to **Settings** > **Kubernetes**.

   - Check the Enable Kubernetes checkbox and hit the **Apply & Restart** button.



Kubernetes

v1.21.3

☑ Enable Kubernetes
Start a Kubernetes single-node cluster when starting Docker Desktop.

□ Show system containers (advanced)
Show Kubernetes internal containers when using Docker commands.

All stacks and Kubernetes resources will be deleted.

Figure 2.4: Docker desktop kubernetes settings

4. Install Helm using the Installing Helm documentation.

5. Install NGINX ingress using helm.

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update

helm upgrade --install --version=4.0.19 ingress-nginx ingress-nginx/ingress-nginx
```

Running the solution

1. Add entries to the hosts file (if you don't know where it is in the file system, check out the location of the hosts file in your operating system):

```
127.0.0.1 eshop-st-web
127.0.0.1 eshop-st-public-web
127.0.0.1 eshop-st-authserver
127.0.0.1 eshop-st-identity
127.0.0.1 eshop-st-administration
127.0.0.1 eshop-st-basket
127.0.0.1 eshop-st-catalog
127.0.0.1 eshop-st-ordering
127.0.0.1 eshop-st-payment
127.0.0.1 eshop-st-gateway-web
127.0.0.1 eshop-st-gateway-web-public
```

2. Execute this etc/scripts/build-images.ps1 script using the powershell.
3. Run this etc/k8s/deploy-staging.ps1 script using the powershell. It will be deployed with the eshop namespace. *You may wait ~30 seconds on the first run to prepare the database.*
4. Browse https://eshop-st-public-web for public and https://eshop-st-web for web application.
    - Username: admin, Password: 1q2w3E*.

Running on HTTPS

You can also run the staging solution on your local Kubernetes cluster with https. There are various ways to create a self-signed certificate.

Installing mkcert

This guide will use mkcert to create self-signed certificates. Follow the installation guide to install mkcert.

Creating mkcert Root CA

Use this command to create root (local) certificate authority for your certificates:

```
mkcert -install
```

Note: all the certificates created by mkcert certificate creation will be trusted by the local machine

Run mkcert

Create certificate for the eshopOnAbp domains using the mkcert command below:

```
mkcert "eshop-st-web" "eshop-st-public-web" "eshop-st-authserver" "eshop-st-identity" "eshop-st-administration" "eshop-st
```

At the end of the output you will see something like:

The certificate is at ./eshop-st-web+10.pem and the key is at ./eshop-st-web+10-key.pem

Copy the cert name and key name below to create TLS secret.

```
kubectl create namespace eshop
kubectl create secret tls -n eshop eshop-wildcard-tls --cert=./eshop-st-web+10.pem  --key=./eshop-st-web+10-key.pem
```

Summary

Completing this section shows you to build and run the solution in a couple of ways. Choose best way that fits you and use that method in next chapters.

Chapter 3: Understanding the Authentication System

This chapter explains the authentication and authorization system in general and how it's used in the solution. It explains the differences between authentication and authorization with real-world samples, discusses the authentication flows and the single sign-on system, points the IdentityServer configuration and the clients.

IdentityServer4 is an OpenId Certified, open-source OpenID Connect and OAuth 2.0 framework for ASP.NET Core. It is considered to be the de-facto standard for authentication in distributed systems. Even Microsoft is using IdentityServer for the default Blazor and SPA application templates. While this book is being written, IdentityServer5 was announced with a different license model by the newly formed Duende Software Inc. The future license may be subject to changes. However, the eShopOnAbp sample will be using IdentityServer4 until further announcements.

Understanding the Differences Between Authentication and Authorization

> Authentication is the movie ticket that allows you to enter cinema and the authorization is the seat where you are allowed to sit.

This phrase is a well-known metaphor to explain the difference between authentication and the authorization.

Figure 1.1: Movie ticket with seat information

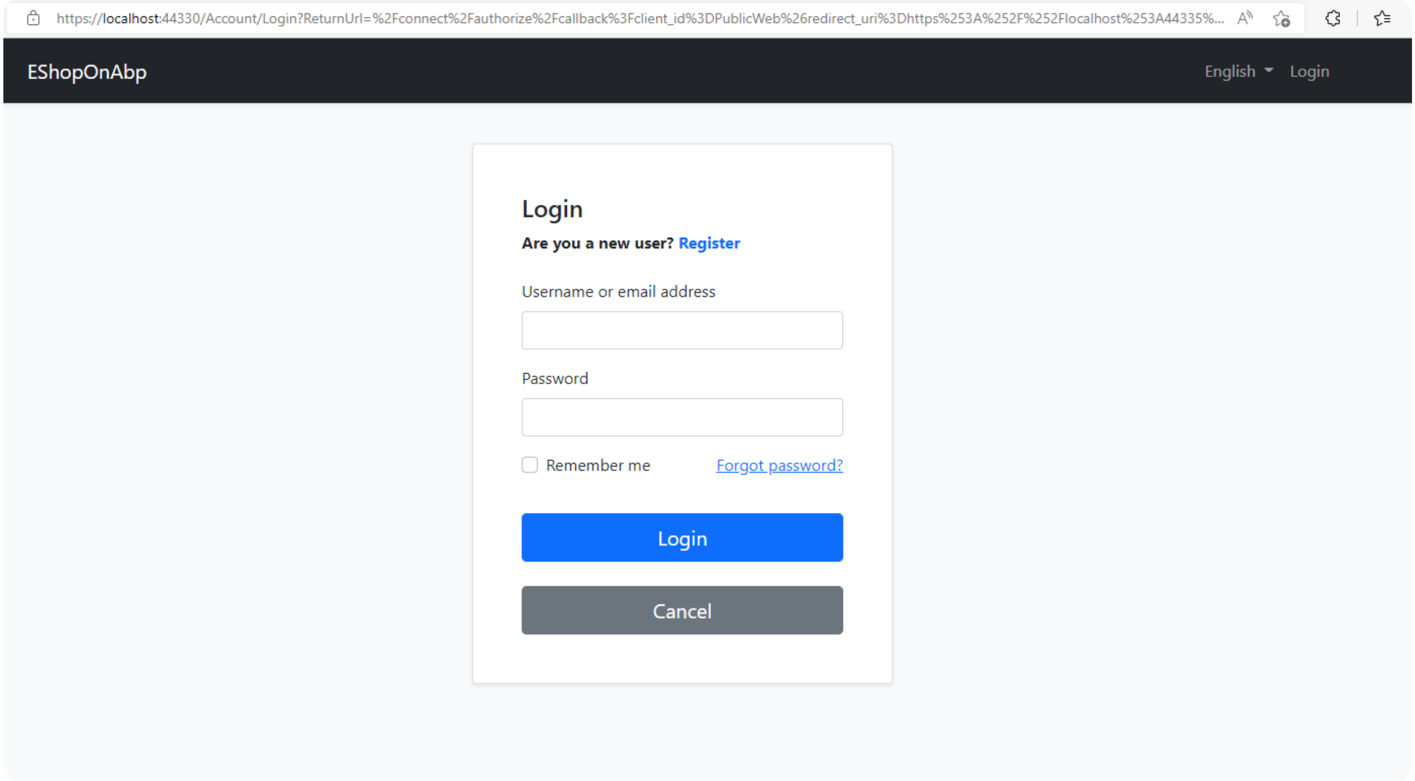
In order to elaborate the concept in the software development world, applications contain various kinds of data presented on a web page. While part of the data can be viewed **anonymously**, some of the data may require extra information about the viewer. The information requirement can be just a registration to the application. In some cases, registration may not be enough alone and the user may need to have more specific information to reach a protected data or perform a specific operation. This information can be a user-specific role, permission, policy, or operation information **granted** by the application authorization administrator. For a real-world application sample, a car company can have a back-office application used mainly by the company employees. This **back-office application** makes in-company operations such as updating product information, profit/loss analysis reporting or tracking employee vacations. It is very likely that the same car company can also have a **landing-page application** that promotes their new cars, side products or marketing announcements. Such applications most likely do not require any kind of registration process to reach a wide range of people since visitors will not want to lose time on complicated registration processes for no good reason.

Although, the car company may have **multiple** back-office or landing-page applications. An employee may be required to use different applications for different tasks. To use different applications, the employee needs to register to all the applications individually and remember all the passwords and upload the same profile pictures. Also, better not forget to logout from each application if they were using a public PC. There should be a better way already.

Understanding Single Sign-on and Single Logout

Instead of registering a user to different applications repeatedly, we can register the user once and all the related independent applications can use this data to authenticate the user across the multiple applications. This authentication method is called single sign-on and it has a complicated implementation directives because of security reasons. Luckily enough, IdentityServer already implements this feature and it is one of the main reasons of using the IdentityServer library.

The eShopOnAbp solution has both a landing-page application (PublicWeb) and a back-office application (angular) for better demonstration of a real-world application sample. Also, the IdentityServer is implemented by the AuthServer (hosted on https://localhost:44330) application. **The Landing-page application** displays the products entered by a back-office application for anonymous users to view and add to their shopping cart. When the purchase process begins, **the user is redirected to the authentication server** (AuthServer) for authentication to select a payment method and a delivery address to complete the process (Figure 1.2).



https://localhost:44330/Account/Login?ReturnUrl=%2Fconnect%2Fauthorize%2Fcallback%3Fclient_id%3DPublicWeb%26redirect_uri%3Dhttps%253A%252F%252Flocalhost%253A44335%...

Login

Are you a new user? Register

Username or email address

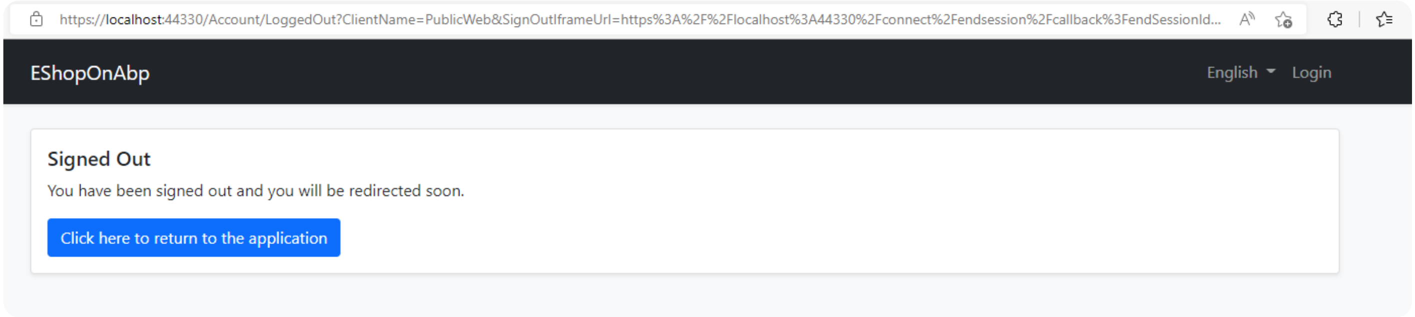
Password

□ Remember me

Figure 1.2: Redirection to Login Page when trying to complete the purchase.

The back-office application contains either management for users, roles, permissions and application settings etc. or product operations like adding, updating, or deleting a product. In order to execute these operations, the user is also required to be authenticated to the back-office application. When using the same browser, if you have already signed in to the back-office application and try to login to the landing-page application, you will immidiately login **without getting asked for your user credentials** because of AuthServer having the shared *idsrv.session* cookie. After the user is successfully logged in, AuthServer grants the user with an access_token like a movie ticket (Figure 1.1).

Other than single sign-in, AuthServer also implements the **single logout** functionality which enables a user to log out of all participating applications in a created session nearly simultaneously (Figure 1.3).


https://localhost:44330/Account/LoggedOut?ClientName=PublicWeb&SignOutIframeUrl=https%3A%2F%2Flocalhost%3A44330%2Fconnect%2Fendsession%2Fcallback%3FendSessionId...

Signed Out

You have been signed out and you will be redirected soon.

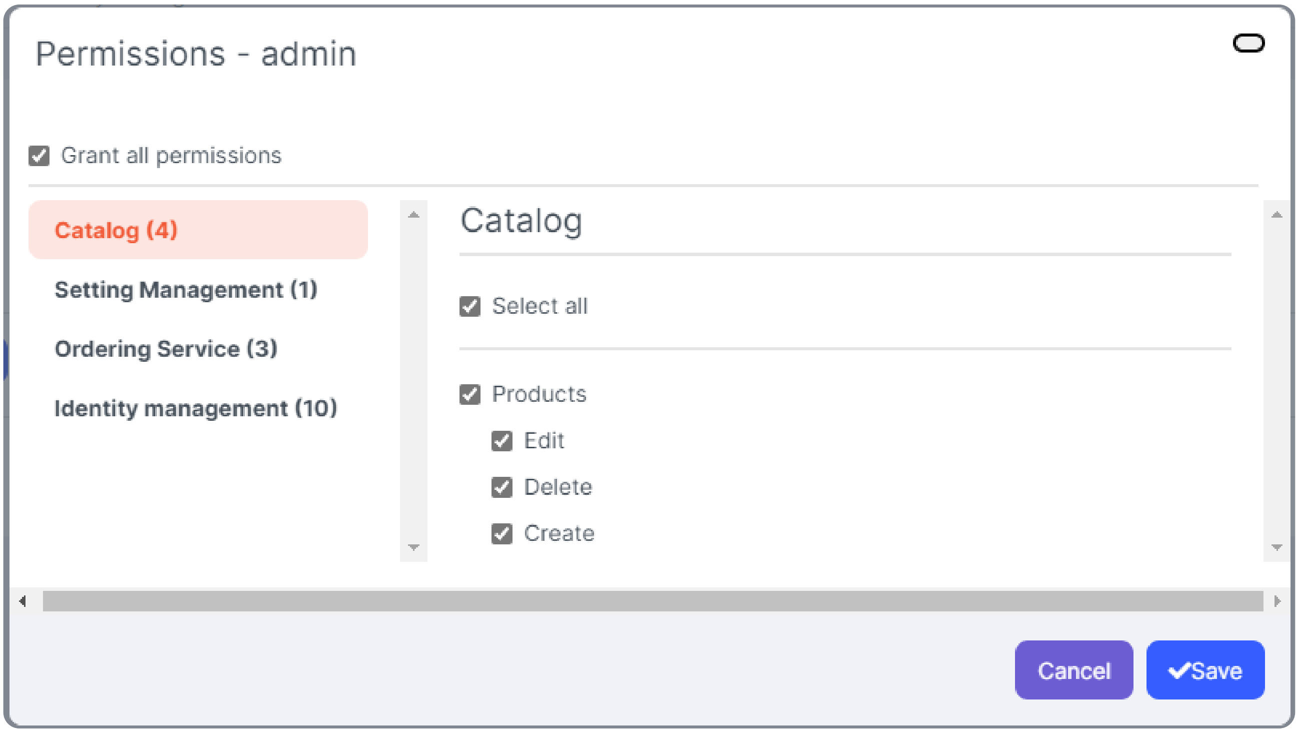
Figure 1.3: Signed Out view after logout is initiated.

When the logout process is initiated, the user is informed with a *Signed Out View* which redirects back to the application after a few seconds later. This is another feature of IdentityServer and it is configured when an IdentityServer client is created.

We have talked about single sign-in where a back-office user can login to the landing-page application because AuthServer uses the same user-store for authentication. But this works both ways that a user signed in to the landing-page application can login to the back-office application too! Does it mean this user can also add new products?

Understanding In-application Authorization

The back-office application contains managements for either users, roles, permissions and application settings etc. or product operations like adding, updating, or deleting a product. All these different kind of operations require different kinds of authorization information after completing an authentication process shown in *Figure 1.1*. The eShopOnAbp benefits from the ABP Permission Management authorization system to handle this complexity by **defining permissions** for each specific action. We define these permissions in the respected microservice. For example, CatalogService already defines the product related create, update, delete permissions as shown in *Figure 1.3*. A back-office user needs to be granted the *Product.Create* permission in order to create a product or to even see the product creation button.



Permissions - admin

☑ Grant all permissions

Catalog (4)

Setting Management (1)

Ordering Service (3)

Identity management (10)

Catalog

☑ Select all

☑ Products

☑ Edit

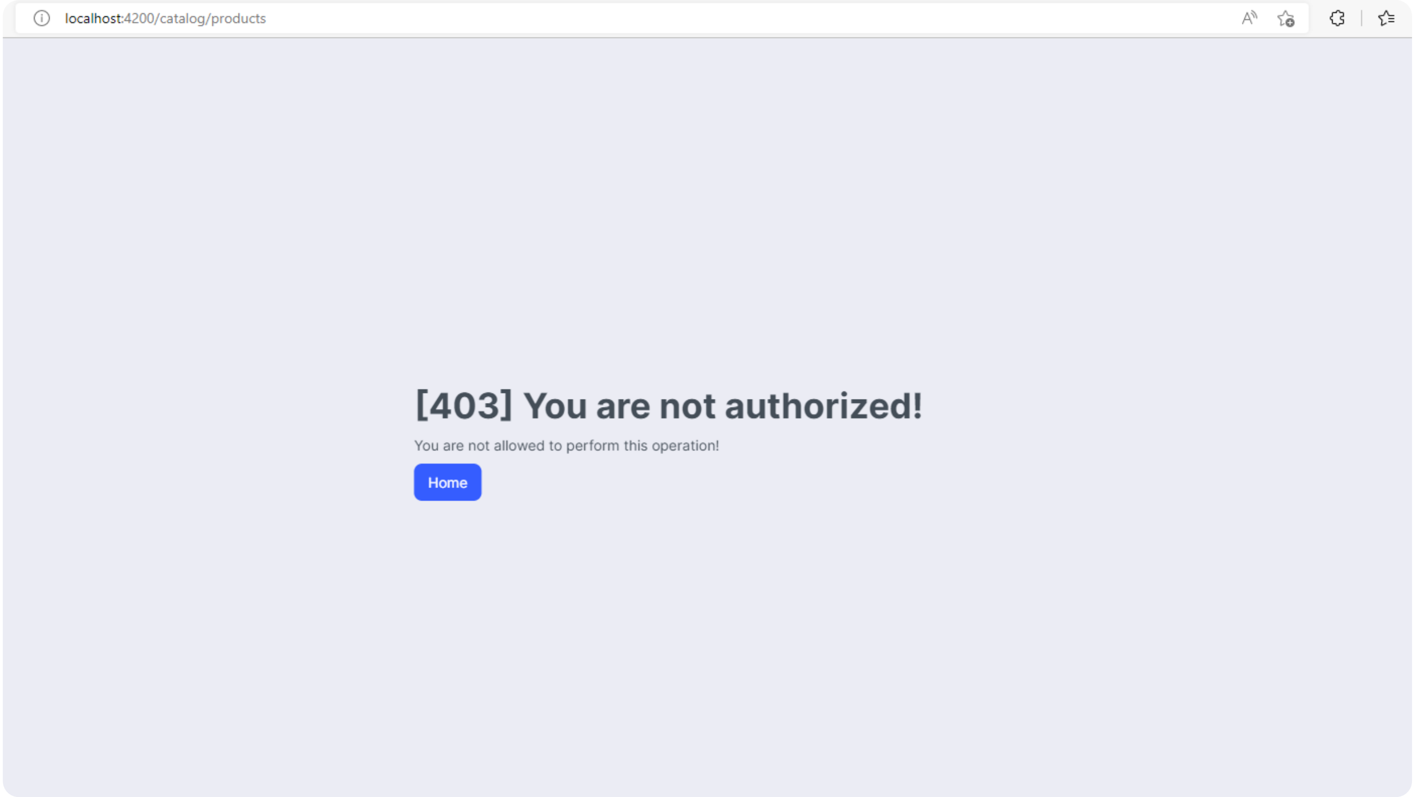
☑ Delete

☑ Create

Figure 1.3: Catalog Service products permission list.

When making an http request to **get the list of products** from the Catalog microservice, the access*token received from the AuthServer is added as the _Bearer* token to the request authorization header. The Catalog microservice verifies the validity of the token and checks if the requested endpoint (application service method) requires extra permission. If the requested resource requires extra permission, ABP Permission Management checks the granted permission list of the logged in user. If the user has the required permission, a request is granted and the product list will be returned by the CatalogService.

The user without related permissions won't be able to see any authorized back-office application pages. If it tries to navigate to a protected page, the user will encounter the unauthorized page as shown in Figure 1.4. If it tries to make an http request to an endpoint, it will result in HTTP **403 Forbidden**.



[403] You are not authorized!

You are not allowed to perform this operation!

Figure 1.4: User trying to reach products list page without permission.

A user logged in to the back-office application can see and operate in pages based on granted permissions. Although, the back-office **application itself** needs to be authorized to make http requests to the necessary microservices in order to get or save data. eShopOnAbp contains numerous microservices and applications. It is important to distinguish **between** the **authorization inside an application** and the **authorization between the applications**. To understand the authorization between applications, we need to be familiar with the IdentityServer terminology.

Understanding the IdentityServer Terminology

In authentication and authorization context, **OpenID** is about authentication (like providing who you are), **OAuth** is about the authorization of a client to access protected resources. However, **OpenID Connect (OIDC)** is an open authentication protocol that profiles and extends OAuth 2.0 to add an identity layer. For a better user-friendly explanation, the IdentityServer provides a secure environment between the applications inside the distributed systems by the certified implementation of OpenID Connect (OIDC). *Authorization of applications* is a wide-range and continuously updated security concept within its own terminology.

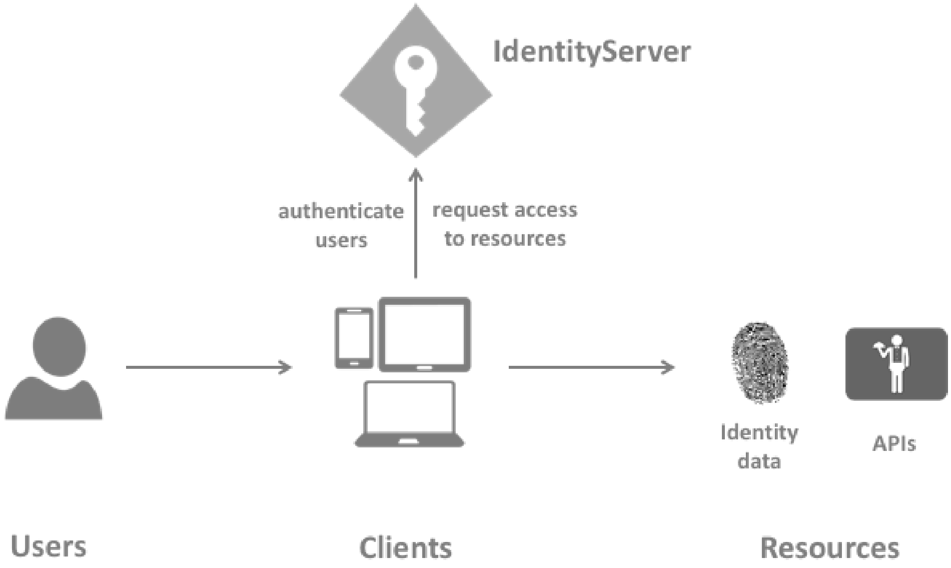


Figure 1.5: IdentityServer terminology from IdentityServer4 docs

A **user** is a human that is using a registered client to access resources.

A **client** is a piece of software that requests tokens from IdentityServer - either for authenticating a user (requesting an identity token) or for accessing a resource (requesting an access token). A client must be first registered with IdentityServer before it can request tokens. Examples for clients are web applications, native mobile, desktop applications, SPAs or server processes.. etc.

Resources are something you want to protect with IdentityServer - either identity data of your users or APIs. Every resource has a unique name and clients use this name to specify which resources they want to get access to. **Identity data** is the identity information (aka claims) about a user, e.g. name or email address. **APIs** are resources that represent a functionality that a client wants to invoke - typically modelled as Web APIs, but not necessarily.

Scopes are parts or a single part of a resource. They can either be Identity scopes like *email*, *given_name* etc. or API scopes that represent a part of your (or your whole) api resource like *myapi.write*, *myapi.read* etc.

An **identity token** represents the outcome of an authentication process. It contains (at a bare minimum) an identifier for the user (called the sub or subject claim) and information about how and when the user is authenticated. It can contain additional identity data.

An **access token** allows access to an API resource. Clients request access tokens and forward them to the API. Access tokens contain information about the client and the user (if present). APIs use that information to authorize access to their data.

You can check IdentityServer4 documentation for detailed explanation of the Figure1.5.

Understanding Authorization Between Applications

In the *authentication server* context, a user login process is an **OpenId Connect flow** of a client which indicates the process of authentication and the authorization of an application to the authentication server ending with an **access_token** (and an identity*token) yield. The flow specs are defined by the Internet Engineering Task Force (IETF) and the implementation details are provided by _openid.net*. The flow varies based on the application types like server-side, SPA, mobile, console application, etc. While detailed explanations of all the OIDC flows are not the focus of this book, we will mention the used OIDC flows in the applications with reasons.

The eShopOnAbp solution contains multiple applications, gateways, and microservices. **The PublicWeb** (landing page) is a server-side rendered by a Razor/MVC application that uses **hybrid flow** which is the suggested authentication flow for these kinds of applications by openid.net. This configuration can be found when adding authentication configuration under the ConfigureServices method of the **EShopOnAbpPublicWebModule**:

```
.AddAbpOpenIdConnect("oidc", options =>
{
    options.Authority = configuration["AuthServer:Authority"];
    options.RequireHttpsMetadata = Convert.ToBoolean(configuration["AuthServer:RequireHttpsMetadata"]);
    options.ResponseType = OpenIdConnectResponseType.CodeIdToken; // Indicates the flow - Hybrid flow

    options.ClientId = configuration["AuthServer:ClientId"];
    options.ClientSecret = configuration["AuthServer:ClientSecret"];

    options.SaveTokens = true;
    options.GetClaimsFromUserInfoEndpoint = true;

    options.Scope.Add("role");
    options.Scope.Add("email");
    options.Scope.Add("phone");
    options.Scope.Add("AccountService");
    options.Scope.Add("AdministrationService");
    options.Scope.Add("BasketService");
    options.Scope.Add("CatalogService");
    options.Scope.Add("PaymentService");
    options.Scope.Add("OrderingService");
});
```

The Angular (back-office) is an SPA application written with angular that uses **Authorization Code Flow with PKCE** which becomes the suggested flow after the deprecation of the implicit flow. This is configured at oAuthConfig located in **environment.ts** file:

```
oAuthConfig: {
  issuer: 'https://localhost:44330',
  redirectUri: baseUrl,
  clientId: 'Web',
  responseType: 'code', // Indicates the flow - Authorization Code flow
  scope: 'offline_access openid profile email phone AccountService IdentityService AdministrationService Catalo
 },
```

Authorization code flow redirects the user to AuthServer for the login process. In some cases like porting the SPA application to a mobile application, the SPA application may want to host the login page and require the user to enter the username and password to the SPA application. Even if it's not recommended, if the SPA application is trusted by the AuthServer, this behavior can be achieved by using the **resource owner password credentials** flow. To change the angular application *authorization code* flow to the *resource owner password credentials* flow, replace the responseType **code** with **password**. Or completely remove the responseType line where the ABP account module **falls back** to default password response type. After making this change, when trying to login from the angular application, users will encounter the login page hosted on the angular application (Figure 1.6).

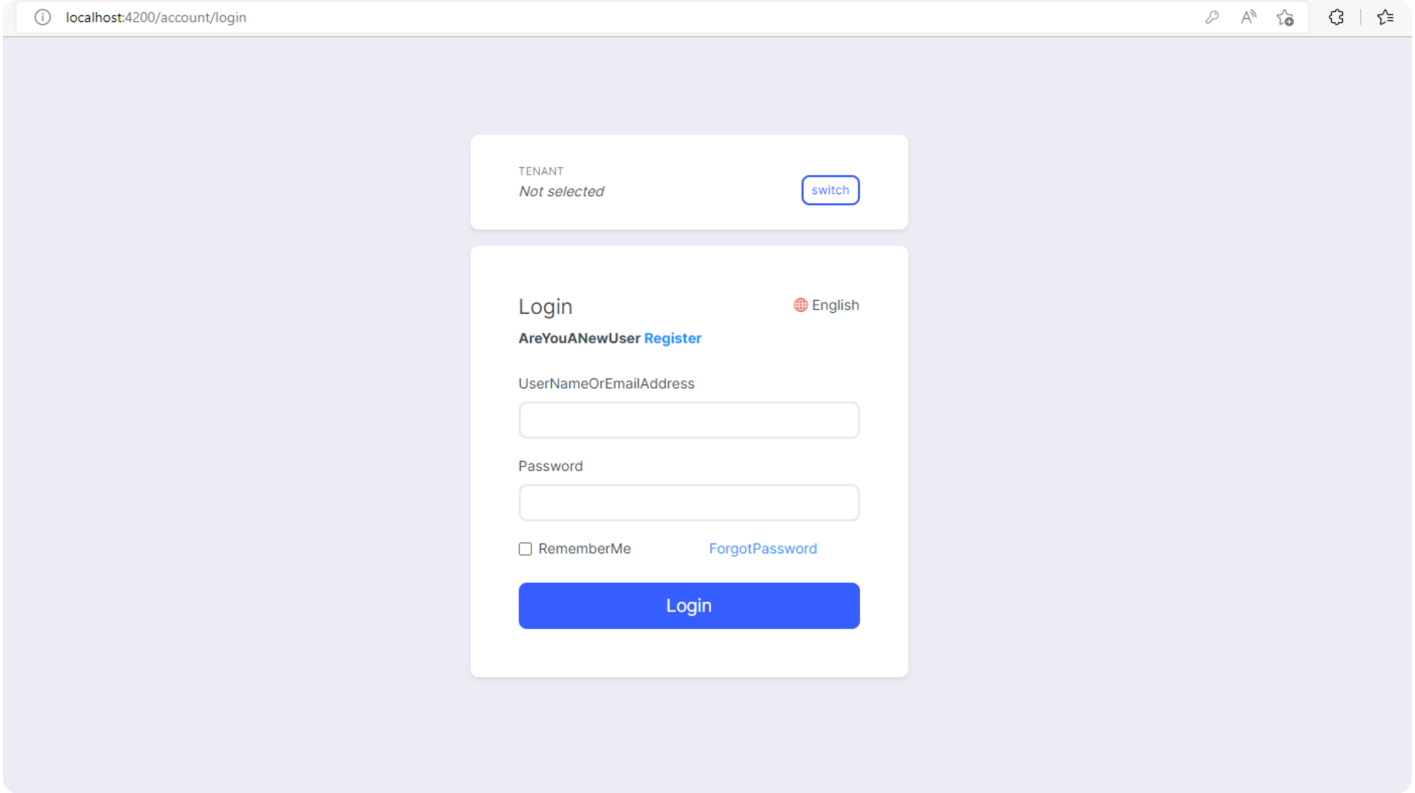


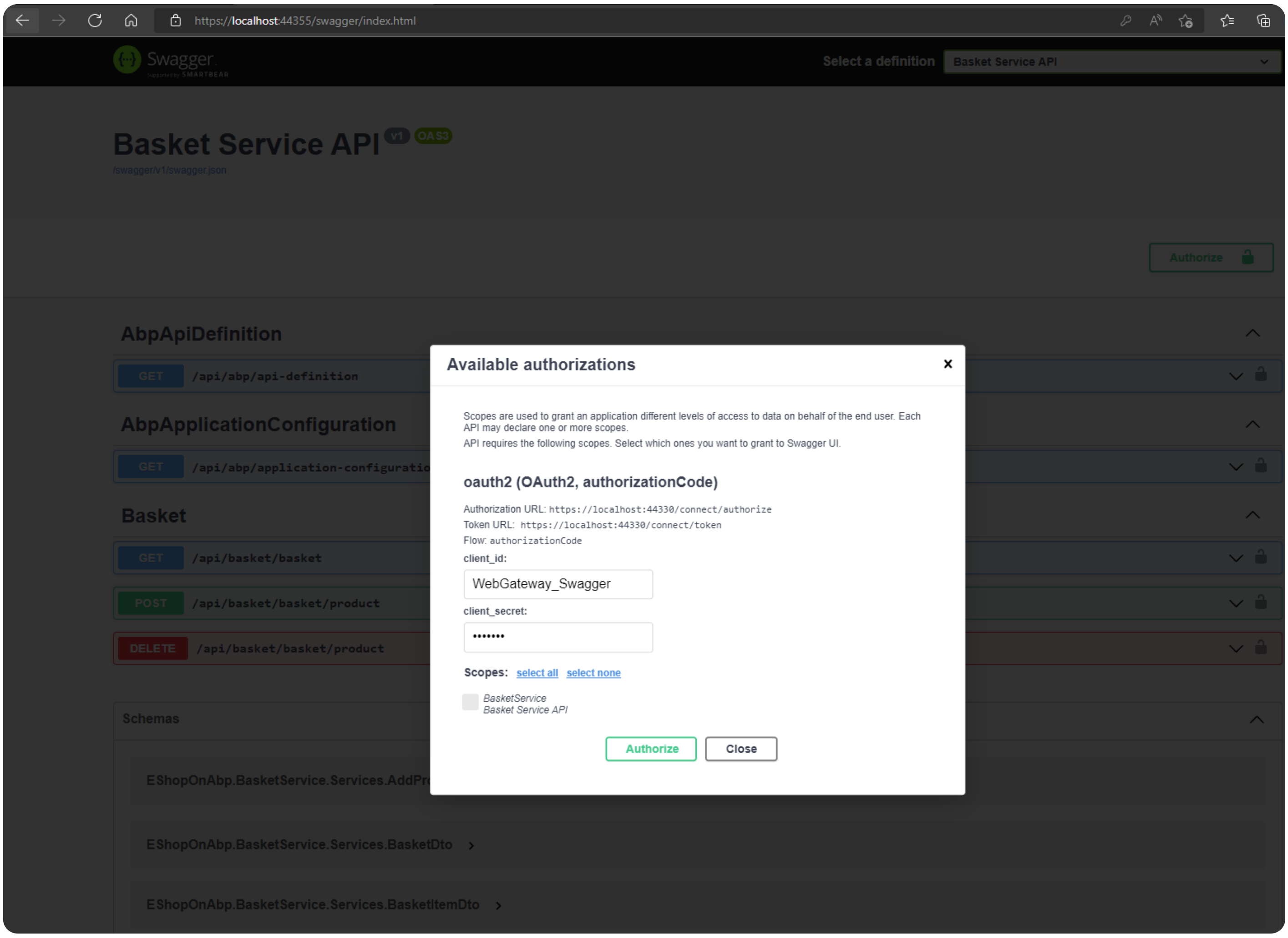
Figure 1.6: Angular (Back-Office) application using Resource Owner Password Credentials flow

Keep in mind that, to make the authorization flow work for a client, the client should be **allowed to use** that grant type by the AuthServer. For example, the angular application is granted to use both *authorization code* and *resource owner password credentials* by the AuthServer (see details in the Configuring IdentityServer section). Otherwise, it would result in an error like Invalid grant type for client: password in the authorization process.

The **WebGateway_Swagger** client is used for swagger authorization using the Volo.Abp.Swashbuckle library for authorization on Swagger UI. This configuration is done at SwaggerConfigrationHelper under *EShopOnAbp.Shared.Hosting.AspNetCore* shared project that makes swagger authorization available for all the dependent projects. The resource configuration for the BasketService is as follows:

```
SwaggerConfigurationHelper.ConfigureWithAuth(
    context: context,
    authority: configuration["AuthServer:Authority"],
    scopes: new
    Dictionary<string, string> /* Requested scopes for authorization code request and descriptions for swagger UI only */
    {
        {"BasketService", "Basket Service API"}
    },
    apiTitle: "Basket Service API"
);
```

This shared configuration adds the **Authorize button** with authorization code flow which redirects to the AuthServer to authorize the user to the swagger application using **client_id** and **client_secret** with a selected scope request then redirects back to Swagger UI. Since the client_secret is sensitive data to use on browser, it is automatically filled by the AbpSwashbuckleModule. You can see the authorization modal for Basket microservice Swagger UI on Figure 1.7.


https://localhost:44355/swagger/index.html
Swagger
Select a definition
Basket Service API
Basket Service API
v1
OAS3
/swagger/v1/swagger.json
Authorize
AbpApiDefinition
GET /api/abp/api-definition
AbpApplicationConfiguration
GET /api/abp/application-configuratio
Basket
GET /api/basket/basket
POST /api/basket/basket/product
DELETE /api/basket/basket/product
Schemas
EShopOnAbp.BasketService.Services.AddPr
EShopOnAbp.BasketService.Services.BasketDto
EShopOnAbp.BasketService.Services.BasketItemDto

Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.

API requires the following scopes. Select which ones you want to grant to Swagger UI.

oauth2 (OAuth2, authorizationCode)

Authorization URL: `https://localhost:44330/connect/authorize`

Token URL: `https://localhost:44330/connect/token`

Flow: `authorizationCode`

client_id: WebGateway_Swagger

client_secret: •••••••

Scopes: select all select none

□ *BasketService*
Basket Service API

Figure 1.7: Basket Service API Swagger Authorization

This client has all the scope, redirect URI, and CORS configurations for **accessing** all the API resources. The flow exchanges code for access tokens using client id and secret after the user logs in. But how to authenticate non-user involved applications? Is that a thing? Do we ever use or need it?

> A bar asks a cinema how many tickets did it sell

This phrase may sound dumb and irrational but that is pretty much what happens when a *service-to-service call* occurs in distributed systems. In this concept, the service is a back-end application making an HTTP (or gRPC) request to another back-end application. This is widely seen in inter-communication between microservices.

To achieve security in-between server-to-server interactions, **the Client credentials flow** is the suggested approach. There is **no user involved** in the process and resulting access token will not contain a user but a client id. A sample of this kind of interaction happens when the **user permissions are requested** on the Identity management administration pages. While the Permission management is hosted by the AdministrationService, the user (identity) list is hosted by the IdentityService which is required to show the user permissions. In this case, **AdministrationService is a client** that makes a request to the **IdentityService resource**. While this process requires steps of requests to AuthServer for the access token, Volo.Abp.Http.Client.IdentityModel.Web library abbreviates the process by just adding configuration to the appsettings file of the client. The AdministrationService client credentials flow configuration in appsettings.json is as follows:

```
"IdentityClients": {
  "Default": {
    "GrantType": "client_credentials",
    "ClientId": "EShopOnAbp_AdministrationService",
    "ClientSecret": "1q2w3e*",
    "Authority": "https://localhost:44330",
    "Scope": "IdentityService"
  }
 },
```

The authorization **between the applications** is achieved by the AuthServer OIDC protocol but the requested IdentityService resource has an extra permission requirement and there is **no user involved to have a permission**. The ABP Permission Management system solves this problem by granting **permissions to clients** (with a "C" provider name) like it does to roles and users with different provider names. We can easily make this configuration when configuring the IdentityServer.

Configuring the IdentityServer

While IdentityServer is hosted under AuthServer, the related data and application services are hosted by the IdentityService. To run eShopOnAbp successfully, the distributed environment data must be seeded and that includes the resources, the scopes and the clients. This information can be found on **IdentityServerDataSeeder** which is located inside the *DbMigrations* folder in the *IdentityService.HttpApi.Host project*.

The API resources in the eShopOnAbp solution contain all the microservices, which are IdentityService, AdministrationService, CatalogService, BasketService, OrderingService and the PaymentService. However, apart from microservices, AuthServer hosts the account management module whose endpoints are protected by registering the AuthServer as an API resource named **AccountService**.

```
private async Task CreateApiResourcesAsync()
{
    var commonApiUserClaims = new[] // Identity resources
    {
        "email",
        "email_verified",
        "name",
        "phone_number",
        "phone_number_verified",
        "role"
    };

    await CreateApiResourceAsync("AccountService", commonApiUserClaims);
    await CreateApiResourceAsync("IdentityService", commonApiUserClaims);
    await CreateApiResourceAsync("AdministrationService", commonApiUserClaims);
    await CreateApiResourceAsync("CatalogService", commonApiUserClaims);
    await CreateApiResourceAsync("BasketService", commonApiUserClaims);
    await CreateApiResourceAsync("OrderingService", commonApiUserClaims);
    await CreateApiResourceAsync("PaymentService", commonApiUserClaims);
}
```

The API scopes do not contain a part of the API resource since eShopOnAbp **doesn't use the scope based authorization**. Instead, each API resource is registered as an API scope to be available for the clients to request.

```
private async Task CreateApiScopesAsync()
{
    await CreateApiScopeAsync("AccountService");
    await CreateApiScopeAsync("IdentityService");
    await CreateApiScopeAsync("AdministrationService");
    await CreateApiScopeAsync("CatalogService");
    await CreateApiScopeAsync("BasketService");
    await CreateApiScopeAsync("OrderingService");
    await CreateApiScopeAsync("PaymentService");
}
```

The Clients are defined specifically, based on their application type and most of them contain one or more **redirect URIs** which are defined in the **IdentityServerClients** section of the appsettings.json:

```
"IdentityServerClients": {
  "Web": {
    "RootUrl": "http://localhost:4200"
  },
  "PublicWeb": {
    "RootUrl": "https://localhost:44335/"
  },
  "WebGateway": {
    "RootUrl": "https://localhost:44372"
  },
  "PublicWebGateway": {
    "RootUrl": "https://localhost:44373"
  },
  "AccountService": {
    "RootUrl": "https://localhost:44330"
  },
  "IdentityService": {
    "RootUrl": "https://localhost:44351"
  },
  "AdministrationService": {
    "RootUrl": "https://localhost:44353"
  },
  "CatalogService": {
    "RootUrl": "https://localhost:44354"
  },
  "BasketService": {
    "RootUrl": "https://localhost:44355"
  },
  "OrderingService": {
    "RootUrl": "https://localhost:44356"
  },
  "PaymentService": {
    "RootUrl": "https://localhost:44357"
  }
 }
```

The **PublicWeb client** is configured with *hybrid* grant type with the following definitions:

```
var commonScopes = new[] // Identity scopes
{
    "email",
    "openid",
    "profile",
    "role",
    "phone",
    "address"
};

//Public Web Client
var publicWebClientRootUrl = _configuration["IdentityServerClients:PublicWeb:RootUrl"]
    .EnsureEndsWith('/');
await CreateClientAsync(
    name: "PublicWeb",
    scopes: commonScopes.Union(new[]
                              {
                                  "AccountService",
                                  "AdministrationService",
                                  "CatalogService",
                                  "BasketService",
                                  "PaymentService",
                                  "OrderingService"
                              }),
    grantTypes: new[] { "hybrid" },
    secret: "1q2w3e*".Sha256(),
    redirectUris: new List<string> { $"{publicWebClientRootUrl}signin-oidc" },
    postLogoutRedirectUri: $"{publicWebClientRootUrl}signout-callback-oidc",
    frontChannelLogoutUri: $"{publicWebClientRootUrl}Account/FrontChannelLogout",
    corsOrigins: new[] { publicWebClientRootUrl.RemovePostFix("/") }
);
```

The **Angular client** is configured with *authorization_code*, *password* and the custom *LinkLogin* flow type which allows user impersonation with the following definitions:

```
//Angular Client
var angularClientRootUrl =_configuration["IdentityServerClients:Web:RootUrl"].TrimEnd('/');
await CreateClientAsync(
    name: "Web",
    scopes: commonScopes.Union(new[]
                            {
                                "AccountService",
                                "IdentityService",
                                "AdministrationService",
                                "CatalogService",
                                "OrderingService"
                            }),
    grantTypes: new[] { "authorization_code", "LinkLogin", "password" },
    secret: "1q2w3e*".Sha256(),
    requirePkce: true,
    requireClientSecret: false,
    redirectUris: new List<string> { $"{angularClientRootUrl}" },
    postLogoutRedirectUri: $"{angularClientRootUrl}",
    corsOrigins: new[] { angularClientRootUrl }
);
```

The **Swagger client** is configured with the *authorization_code* flow and is used in all the microservices, gateways and AccountService (AuthServer) with the following definitions:

```
// Swagger Client
var swaggerClientId = $"{name}_Swagger";
if (!swaggerClientId.IsNullOrWhiteSpace())
{
    var webGatewaySwaggerRootUrl = _configuration[$"IdentityServerClients:{name}:RootUrl"].TrimEnd('/');
    var publicWebGatewayRootUrl = _configuration[$"IdentityServerClients:PublicWebGateway:RootUrl"].TrimEnd('/');
    var accountServiceRootUrl = _configuration[$"IdentityServerClients:AccountService:RootUrl"].TrimEnd('/');
    var identityServiceRootUrl = _configuration[$"IdentityServerClients:IdentityService:RootUrl"].TrimEnd('/');
    var administrationServiceRootUrl = _configuration[$"IdentityServerClients:AdministrationService:RootUrl"].TrimEnd('/');
    var catalogServiceRootUrl = _configuration[$"IdentityServerClients:CatalogService:RootUrl"].TrimEnd('/');
    var basketServiceRootUrl = _configuration[$"IdentityServerClients:BasketService:RootUrl"].TrimEnd('/');
    var orderingServiceRootUrl = _configuration[$"IdentityServerClients:OrderingService:RootUrl"].TrimEnd('/');
    var paymentServiceRootUrl = _configuration[$"IdentityServerClients:PaymentService:RootUrl"].TrimEnd('/');

    await CreateClientAsync(
        name: swaggerClientId,
        scopes: commonScopes.Union(scopes),
        grantTypes: new[] { "authorization_code" },
        secret: "1q2w3e*".Sha256(),
        requireClientSecret: false,
        redirectUris: new List<string>
        {
            $"{webGatewaySwaggerRootUrl}/swagger/oauth2-redirect.html", // WebGateway redirect uri
            $"{publicWebGatewayRootUrl}/swagger/oauth2-redirect.html", // PublicWebGateway redirect uri
            $"{accountServiceRootUrl}/swagger/oauth2-redirect.html", // AccountService redirect uri
            $"{identityServiceRootUrl}/swagger/oauth2-redirect.html", // IdentityService redirect uri
            $"{administrationServiceRootUrl}/swagger/oauth2-redirect.html", // AdministrationService redirect uri
            $"{catalogServiceRootUrl}/swagger/oauth2-redirect.html", // CatalogService redirect uri
            $"{basketServiceRootUrl}/swagger/oauth2-redirect.html", // BasketService redirect uri
            $"{orderingServiceRootUrl}/swagger/oauth2-redirect.html", // OrderingService redirect uri
            $"{paymentServiceRootUrl}/swagger/oauth2-redirect.html" // PaymentService redirect uri
        },
        corsOrigins: new[]
        {
            webGatewaySwaggerRootUrl.RemovePostFix("/"),
            publicWebGatewayRootUrl.RemovePostFix("/"),
            accountServiceRootUrl.RemovePostFix("/"),
            identityServiceRootUrl.RemovePostFix("/"),
            administrationServiceRootUrl.RemovePostFix("/"),
            catalogServiceRootUrl.RemovePostFix("/"),
            basketServiceRootUrl.RemovePostFix("/"),
            orderingServiceRootUrl.RemovePostFix("/"),
            paymentServiceRootUrl.RemovePostFix("/")
        }
    );
}
```

The **Administration Service client** is configured with the *client_credentials* flow which is used to get the user list from the IdentityService when requesting user permission:

```
//Administration Service Client
await CreateClientAsync(
    name: "EShopOnAbp_AdministrationService",
    scopes: commonScopes.Union(new[]
                          {
                              "IdentityService"
                          }),
    grantTypes: new[] { "client_credentials" },
    secret: "1q2w3e*".Sha256(),
    permissions: new[] { IdentityPermissions.Users.Default } // Required permission to make a request to IdentityUser endpoin
);
```

Client creation is very important since if you try to request a non-allowed scope, it will result in an error like error: invalid_scope. To prevent this error, make sure that the scope you are requesting has been granted when the client is created.

Troubleshooting the IdentityServer

The suggested way to troubleshoot the errors by the IdentityServer team is using the application logs. Follow the request trace and check the related application logs. It may be related to reaching the AuthServer, validating the token, or something else. Some of the errors you may come across can be **obscured** for security reasons as the following:

Exception occurred while processing message.
System.InvalidOperationException: IDX20803: Unable to obtain configuration from: 'System.String'.

Detailed information about the errors like these is to show the Personally Identifiable Information (**PII**). The eShopOnContainers applications and microservices enable this on the development environment inside the module configuration as follows:

```
Microsoft.IdentityModel.Logging.IdentityModelEventSource.ShowPII = hostingEnvironment.IsDevelopment();
```