



700 XP

Implement feature flags in a cloud-native ASP.NET Core microservices app

42 min • Module 6 Units

Intermediate

Developer

Solution Architect

ASP.NET Core

Azure

Azure App Configuration

Implement a feature flag in your cloud-native ASP.NET Core microservices app to enable or disable a feature in real time.

Learning objectives

In this module, you will:

- Review ASP.NET Core app configuration concepts.
- Implement real-time feature toggling with the .NET Feature Management library.
- Implement a centralized Azure App Configuration store.

[Start >](#)[⊕ Add](#)

Prerequisites

- Familiarity with C# and ASP.NET Core development at the beginner level
- Familiarity with RESTful service concepts at the beginner level
- Access to an Azure subscription with **Owner** privilege
- Ability to run development containers in Visual Studio Code or GitHub Codespaces

This module is part of these learning paths

[Create cloud-native apps and services with .NET and ASP.NET Core](#)

[Introduction](#)

1 min

Review app configuration concepts

12 min

Exercise - Implement a feature flag to control ASP.NET Core app features

12 min

Exercise - Implement configuration settings in Azure App Configuration instance

10 min

Knowledge check

5 min

Summary

2 min

Introduction

1 minute

Imagine you're a software developer for an online retailer. The retailer's online storefront is a cloud-native, microservices-based ASP.NET Core app. You've been asked to add the ability to the app to have seasonal sales. The sales and the discounts need to be controlled by the sales team, so that app can't be recompiled or redeployed to see the changes.

This module guides you through implementing a feature flags library. This library creates a feature flag to toggle the visibility of the seasonal sale. The configuration values that support this feature flag are centralized by using the Azure App Configuration service.

You use your own Azure subscription to deploy the resources in this module. If you don't have an Azure subscription, create a [free account](#) before you begin.

❗ Important

To avoid unnecessary charges in your Azure subscription, be sure to delete your Azure resources when you're done with this module.

Development container

This module includes configuration files that define a [development container](#), or *dev container*. Using a dev container ensures a standardized environment that's preconfigured with the required tools.

The dev container can run in either of two environments. Before you begin, follow the steps in one of the following links to set up your environment, including installing Docker and the necessary Visual Studio Code extensions.

- [Visual Studio Code and a supported Docker environment on your local machine.](#)
- [GitHub Codespaces](#) (costs may apply).

Learning objectives

- Review ASP.NET Core app configuration concepts.

- Implement real-time feature toggling with the .NET Feature Management library.
- Implement a centralized Azure App Configuration store.
- Implement code to use features and configuration settings from the Azure App Configuration store.

Prerequisites

- Familiarity with C# and ASP.NET Core development at the beginner level.
- Familiarity with RESTful service concepts at the beginner level.
- Conceptual knowledge of containers.
- Access to an Azure subscription with **Owner** privilege.
- Ability to run development containers in Visual Studio Code or GitHub Codespaces.

Next unit: Review app configuration concepts

[Continue >](#)

Review app configuration concepts

12 minutes

Creating microservices for a distributed environment presents a significant challenge. Cloud-hosted microservices often run in multiple containers in various regions. Implementing a solution that separates each service's code from configuration eases the triaging of issues across all environments.

In this unit, explore how to integrate ASP.NET Core and Docker configuration features with Azure App Configuration to tackle this challenge in an effective way.

You'll review the:

- ASP.NET Core configuration infrastructure.
- Kubernetes configuration abstraction—the ConfigMap.
- Azure App Configuration service.
- .NET Feature Management library.
- Feature flag components implemented in the app.

ASP.NET Core configuration

Configuration in an ASP.NET Core project is contained in one or more .NET *configuration providers*. A [configuration provider](#) is an abstraction over a specific configuration source, such as a JSON file. The configuration source's values are represented as a collection of key-value pairs.

An ASP.NET Core app can register multiple configuration providers to read settings from various sources. With the default application host, several configuration providers are automatically registered. The following configuration sources are available in the order listed:

1. JSON file (*appsettings.json*)
2. JSON file (*appsettings.{environment}.json*)
3. User secrets
4. Environment variables
5. Command line

Each configuration provider can contribute its own key value. Furthermore, any provider can

override a value from a provider that was registered earlier in the chain than itself. Given the registration order in the preceding list, a `UseFeatureManagement` command-line parameter overrides a `UseFeatureManagement` environment variable. Likewise, a `UseFeatureManagement` key within `appsettings.json` can be overridden by a `UseFeatureManagement` key stored in `appsettings.Development.json`.

Configuration key names can describe a hierarchy. For example, the notation `eShop:Store:SeasonalSale` refers to the **SeasonalSale** feature within the **Store** microservice of the **eShop** app. This structure can also map configuration values to an object graph or an [array](#).

❗ Important

Some platforms don't support a colon in environment variable names. To ensure cross-platform compatibility, a double underscore (`__`) is used instead of a colon (`:`) to delimit keys. For example, `eShop__Store__SeasonalSale` is the cross-platform equivalent notation for `eShop:Store:SeasonalSale`.

ASP.NET Core uses a [ConfigurationBinder](#) to map configuration values to objects and arrays. The mapping to key names occurs in a case-insensitive fashion. For example, `ConnectionString` and `connectionstring` are treated as equivalent keys. For more information, see [keys and values](#).

Docker configuration

In Docker, one abstraction to handle configuration as a key-value pairs collection is the environment variable section of a container's YAML file. The following snippet is an excerpt from the app's `docker-compose.yml` file:

Dockerfile

```
services:

  frontend:
    image: storeimage
    build:
      context: .
      dockerfile: DockerfileStore
    environment:
      - ProductEndpoint=http://backend:8080
      - ConnectionStrings:AppConfig=Endpoint=https://eshop-app-features.azconfig.io;Id=QWQy;Secret=V/4r/rhg/0tdy2L/
```

```
AmMfBUcgTrYC4krRC7uFqbjRvDU=  
  ports:  
    - "32000:8080"  
  depends_on:  
    - backend
```

The file snippet defines:

- Variables stored in the `environment` section of the YAML file, as highlighted in the preceding snippet.
- Presented to the containerized app as environment variables.
- A mechanism to persist .NET configuration values in microservices apps.

Environment variables are a cross-platform mechanism for providing runtime configuration to apps hosted in the Docker containers.

Azure App Configuration

A centralized configuration service is especially useful in microservices apps and other distributed apps. This module introduces Azure App Configuration as a service for centrally managing configuration values—specifically for feature flags. The service eases the troubleshooting of errors that arise when configuration is deployed with an app.

App Configuration is a fully managed service that encrypts key values both at rest and in transit. Configuration values stored with it can be updated in real time without the need to redeploy or restart an app.

In an ASP.NET Core app, Azure App Configuration is registered as a configuration provider. Aside from the provider registration, the app doesn't know about the App Configuration store. Configuration values can be retrieved from it via .NET's configuration abstraction—the `IConfiguration` interface.

Feature Management library

The *Feature Management* library provides standardized .NET APIs for managing feature flags within apps. The library is distributed via NuGet in the form of two different packages named `Microsoft.FeatureManagement` and `Microsoft.FeatureManagement.AspNetCore`. The latter package provides Tag Helpers for use in an ASP.NET Core project's Razor files. The former package is sufficient when the Tag Helpers aren't needed or when not using with an ASP.NET Core project.

The library is built atop `IConfiguration`. For this reason, it's compatible with any .NET configuration provider, including the provider for Azure App Configuration. Because the library is decoupled from Azure App Configuration, integration of the two is made possible via the configuration provider. Combining this library with Azure App Configuration enables you to dynamically toggle features without implementing supporting infrastructure.

Integration with Azure App Configuration

To understand the integration of Azure App Configuration and the Feature Management library, see the following excerpt from an ASP.NET Core project's `Program.cs` file:

C#

```
string connectionString =
builder.Configuration.GetConnectionString("AppConfig");

// Load configuration from Azure App Configuration
builder.Configuration.AddAzureAppConfiguration(options => {
    options.Connect(connectionString)
        .UseFeatureFlags();
});
```

In the preceding code fragment:

- The app's `builder.Configuration` method is called to register a configuration provider for the Azure App Configuration store. The configuration provider is registered via a call to `AddAzureAppConfiguration`.
- The Azure App Configuration provider's behavior is configured with the following options:
 - Authenticate to the corresponding Azure service via a connection string passed to the `Connect` method call. The connection string is retrieved from the `connectionString` variable. The registered configuration sources are made available via `builder.Configuration`.
 - Enable feature flags support via a call to `UseFeatureFlags`.
- The Azure App Configuration provider supersedes all other registered configuration providers because it's registered after any others.

Tip

In an ASP.NET Core project, you can access the registered providers list by analyzing the `configBuilder.Sources` property inside of `ConfigureAppConfiguration`.

Next unit: Exercise - Implement a feature flag to control ASP.NET Core app features

[Continue >](#)

Exercise - Implement a feature flag to control ASP.NET Core app features

12 minutes

In this exercise, implement a feature flag to toggle a seasonal sales banner for your application. Feature flags allow you to toggle feature availability without redeploying your app.

You'll use the **Feature Management** in the .NET feature flag library. This library provides helpers to implement feature flags in your app. The library supports simple use cases like conditional statements to more advanced scenarios like conditionally adding routes or action filters. Additionally, it supports feature filters, which allow you to enable features based on specific parameters. Examples of such parameters include a window time, percentages, or a subset of users.

In this unit, you will:

- Create an Azure App Configuration instance.
- Add a feature flag to the App Configuration store.
- Connect your app to the App Configuration store.
- Amend the application to use the feature flag.
- Change the products page to display a sales banner.
- Build and test the app.

Open the development environment

You can choose to use a GitHub codespace that hosts the exercise, or complete the exercise locally in Visual Studio Code.

To use a **codespace**, create a preconfigured GitHub Codespace with [this Codespace creation link](#).

GitHub takes several minutes to create and configure the codespace. When it's finished, you see the code files for the exercise. The code that's used for the remainder of this module is in the `/dotnet-feature-flags` directory.

To use **Visual Studio Code**, clone the <https://github.com/MicrosoftDocs/mslearn-dotnet->

[cloudnative](#) repository to your local machine. Then:

1. Install any [system requirements](#) to run Dev Container in Visual Studio Code.
2. Make sure Docker is running.
3. In a new Visual Studio Code window open the folder of the cloned repository
4. Press `Ctrl` + `Shift` + `P` to open the command palette.
5. Search: **>Dev Containers: Rebuild and Reopen in Container**
6. Select **eShopLite - dotnet-feature-flags** from the drop down. Visual Studio Code creates your development container locally.

Create an App Configuration instance

Complete the following steps to create an App Configuration instance in your Azure subscription:

1. In the new terminal pane, sign in to the Azure CLI.

Azure CLI

```
az login --use-device-code
```

2. View your selected Azure subscription.

Azure CLI

```
az account show -o table
```

If the wrong subscription is selected, select the correct one using the [az account set](#) command.

3. Run the following Azure CLI command to get a list of Azure regions and the Name associated with it:

Azure CLI

```
az account list-locations -o table
```

Locate a region closest to you and use it in the next step to replace `[Closest Azure region]`

4. Run the following Azure CLI commands to create an App Configuration instance:

Azure CLI

```
export LOCATION=[Closest Azure region]
export RESOURCE_GROUP=rg-eshop
export CONFIG_NAME=eshop-app-features$SRANDOM
```

You need to change the **LOCATION** to an Azure region close to you, for example **eastus**. If you'd like a different name for your resource group or app configuration change the values above.

5. Run the following command to create the Azure Resource Group:

Azure CLI

```
az group create --name $RESOURCE_GROUP --location $LOCATION
```

6. Run the following command to create an App Configuration instance:

Azure CLI

```
az appconfig create --resource-group $RESOURCE_GROUP --name $CONFIG_NAME
--location $LOCATION --sku Free
```

A variation of the following output appears:

JSON

```
{
  "createMode": null,
  "creationDate": "2023-10-31T15:40:10+00:00",
  "disableLocalAuth": false,
  "enablePurgeProtection": false,
  "encryption": {
    "keyVaultProperties": null
  },
  "endpoint": "https://eshop-app-features1168054702.azconfig.io",
  "id": "/subscriptions/7eebce2a-0884-4df2-8d1d-2a3c051e47fe/
resourceGroups/rg-eshop/providers/Microsoft.AppConfiguration/
configurationStores/eshop-app-features1168054702",
  "identity": null,
```

7. Run this command to retrieve the connection string for the App Configuration instance:

Azure CLI

```
az appconfig credential list --resource-group $RESOURCE_GROUP --name
```

```
$CONFIG_NAME --query [0].connectionString --output tsv
```

This string prefixed with `Endpoint=` represents the App Configuration store's connection string.

8. Copy the connection string. You'll use it in a moment.

Store the App Configuration connection string

You'll now add the App Configuration connection string to the application. Complete the following steps:

1. Open the `/dotnet-feature-flags/docker-compose.yml` file.
2. Add a new environment variable at line 13.

yml

```
- ConnectionStrings:AppConfig=[PASTE CONNECTION STRING HERE]
```

The `docker-compose.yml` will resemble the following YAML:

yml

```
environment:
  - ProductEndpoint=http://backend:8080
  - ConnectionStrings:AppConfig=Endpoint=https://eshop-app-features1168054702.azureconfig.io;Id=<ID>;Secret=<Secret value>
```

The preceding line represents a key-value pair, in which `ConnectionStrings:AppConfig` is an environment variable name. In the *Store* project, the environment variables configuration provider reads its value.

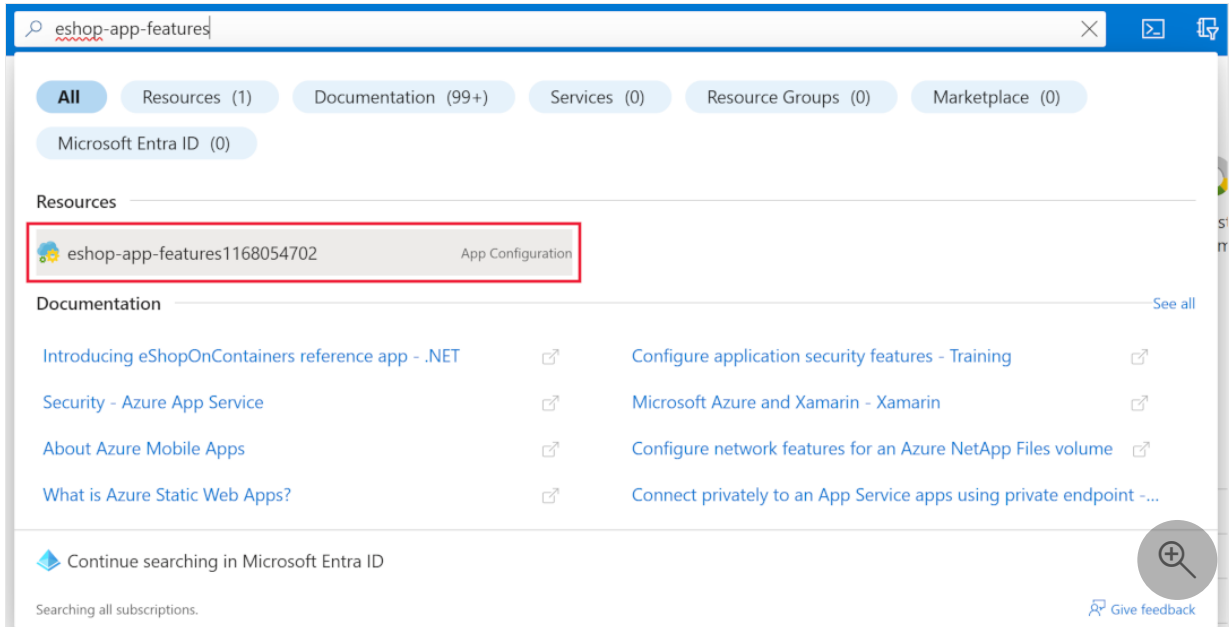
Tip

Your Azure App Configuration connection string contains a plain-text secret. In real world apps, consider integrating App Configuration with Azure Key Vault for secure storage of secrets. Key Vault is out of scope for this module, but guidance can be found at [Tutorial: Use Key Vault references in an ASP.NET Core app](#).

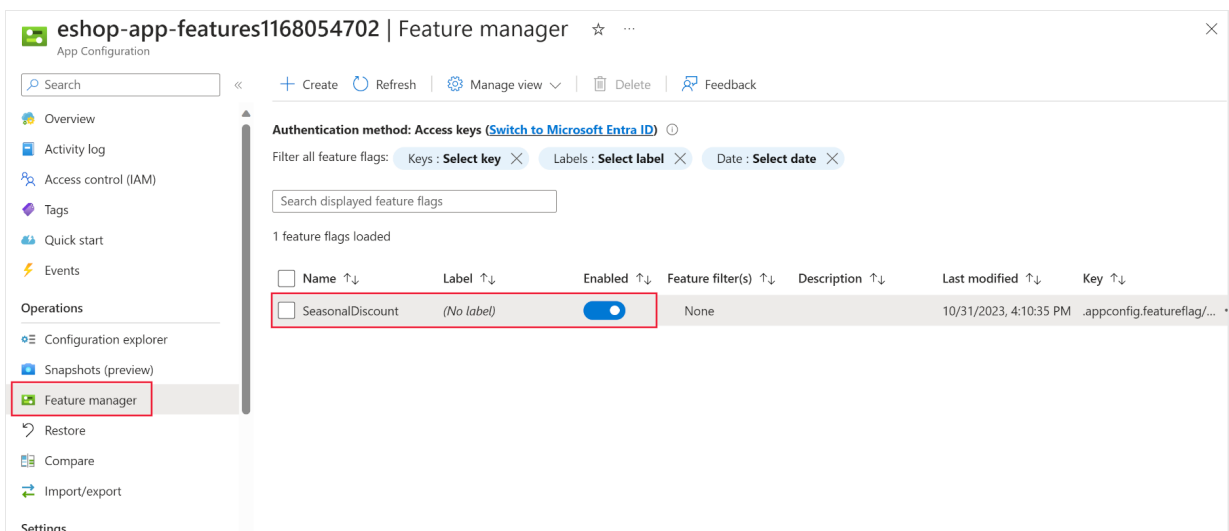
Add the feature flag to the App Configuration store

In Azure App Configuration, create and enable a key-value pair to be treated as a feature flag. Complete the following steps:

1. In another browser tab, sign into the [Azure portal](#) with the same account and directory as the Azure CLI.
2. Use the search box to find and open the App Configuration resource prefixed with **eshop-app-features**.



3. In the **Operations** section, select **Feature manager**.
4. In the top menu, select **+ Create**.
5. Select the **Enable feature flag** check box.
6. In the **Feature flag** name text box, enter **SeasonalDiscount**.
7. Select **Apply**.



Now that the feature flag exists in the App Configuration store, the **Store** project requires some changes to read it.

Review code

Review the directories in the explorer pane in the IDE. Note that there's three projects **DataEntities**, **Products**, and **Store**. The **Store** project is the Blazor app. The **Products** project is a .NET Standard library that contains the product service. The **DataEntities** project is a .NET Standard library that contains the product model.

Connect your app to the App Configuration store

To access values from the App Configuration store in an ASP.NET Core app, the configuration provider for App Configuration is needed.

Apply the following changes to your **Store** project:

1. In the terminal window, navigate to the Store folder:

Bash

```
cd dotnet-feature-flags/Store
```

2. Run the following command to install a NuGet package containing the .NET configuration provider for the App Configuration service:

.NET CLI

```
dotnet add package Microsoft.Azure.AppConfiguration.AspNetCore
dotnet add package Microsoft.FeatureManagement.AspNetCore
dotnet add package
Microsoft.Extensions.Configuration.AzureAppConfiguration
```

3. Open the **Store/Program.cs** file.
4. Add the new package references at the top of the file:

C#

```
using Microsoft.FeatureManagement;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Configuration.AzureAppConfiguration;
```

5. Add this code below the **// Add the AddAzureAppConfiguration code** comment.

```
C#

// Retrieve the connection string
var connectionString =
builder.Configuration.GetConnectionString("AppConfig");

// Load configuration from Azure App Configuration
builder.Configuration.AddAzureAppConfiguration(options => {
    options.Connect(connectionString)
        .UseFeatureFlags();
});

// Register the Feature Management library's services
builder.Services.AddFeatureManagement();
builder.Services.AddAzureAppConfiguration();
```

In the preceding code snippet:

- The `Connect` method authenticates to the App Configuration store. Recall that the connection string is being passed as an environmental variable `ConnectionStrings:AppConfig`.
- The `UseFeatureFlags` method enables the Feature Management library to read feature flags from the App Configuration store.
- The two `builder.Services` calls register the Feature Management library's services with the app's dependency injection container.

6. At the bottom of the file, below **// Add the App Configuration middleware**, add this code:

```
C#

app.UseAzureAppConfiguration();
```

The preceding code adds the App Configuration middleware to the request pipeline. The middleware triggers a refresh operation for the Feature Management parameters for every incoming request. Then it's up to the `AzureAppConfiguration` provider to decide, based on refresh settings, when to actually connect to the store to get the values.

Enable a sales banner

Your app can now read the feature flag, but the products page needs to be updated to

show that a sale is on. Complete the following steps:

1. Open the **Store/Components/Pages/Products.razor** file.
2. At the top of the file, add the following code:

```
C#  
  
@using Microsoft.FeatureManagement  
@inject IFeatureManager FeatureManager
```

The preceding code imports the Feature Management library's namespaces and injects the `IFeatureManager` interface into the component.

3. In the **@code** section, add the following variable to store the state of the feature flag:

```
C#  
  
private bool saleOn = false;
```

4. In the **OnInitializedAsync** method, add the following code:

```
C#  
  
saleOn = await FeatureManager.IsEnabledAsync("SeasonalDiscount");
```

The method should look like the following code:

```
C#  
  
protected override async Task OnInitializedAsync()  
{  
    saleOn = await FeatureManager.IsEnabledAsync("SeasonalDiscount");  
  
    // Simulate asynchronous loading to demonstrate streaming rendering  
    products = await ProductService.GetProducts();  
}
```

5. At line 26, under the **<!-- Add a sales alert for customers -->** comment, add the following code:

```
razor  
  
<!-- Add a sales alert for customers -->  
@if (saleOn)
```

```
{  
<div class="alert alert-success" role="alert">  
  Our sale is now on.  
</div>  
}
```

The preceding code displays a sales alert if the feature flag is enabled.

Build the app

1. Ensure you've saved all your changes, and are in the **dotnet-feature-flags** directory. In the terminal, run the following command:

.NET CLI

```
dotnet publish /p:PublishProfile=DefaultContainer
```

2. Run the app using docker:

Bash

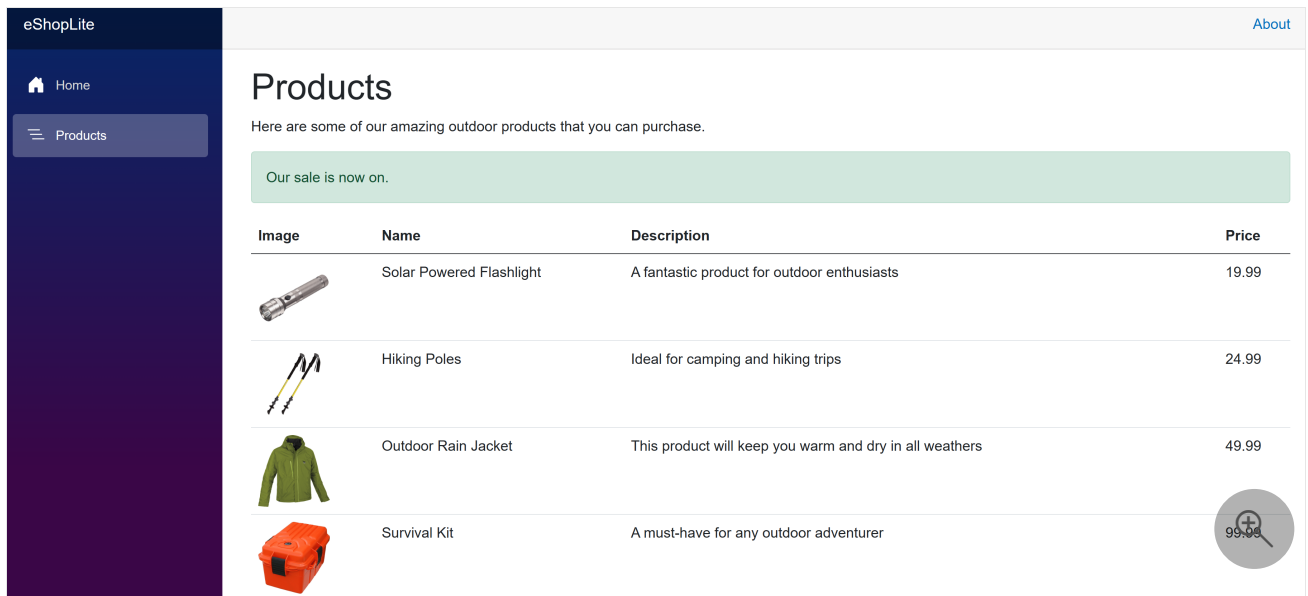
```
docker compose up
```

Test the feature flag

To verify the feature flag works as expected in a codespace, complete the following steps:

1. Switch to the **PORTS** tab, then to the right of the local address for the **Front End** port, select the globe icon. The browser opens a new tab at the homepage.
2. Select **Products**.

If you're using Visual Studio Code locally, open <http://localhost:32000/products>.



In the Azure portal, you can enable and disable the feature flag and refresh the products page to see the flag in action.

Next unit: Exercise - Implement configuration settings in Azure App Configuration instance

[Continue >](#)

Exercise - Implement configuration settings in Azure App Configuration instance

10 minutes

A new requirement to the application now says to allow a feature flag to control a discount for a product. This exercise shows you how to do that.

- Add a configuration setting to the Azure App Configuration instance.
- Add code to use the new setting to discount product prices.
- Build and test the app.

Add the configuration setting to the App Configuration store

In Azure App Configuration, now create a new key-value pair to store the sales discount percentage. Complete the following steps:

1. In another browser tab, sign into the [Azure portal](#) with the same account and directory as the Azure CLI.
2. Use the search box to find and open the App Configuration resource prefixed with **eshop-app-features**.
3. In the **Operations** section, select **Configuration explorer**.
4. In the top menu, select **+ Create** and select **Key-value**.
5. In the **Key** text box, enter **eShopLite__Store__DiscountPercent**.
6. In the **Value** text box, enter **0.8**.
7. Select **Apply**.

Add code to use the new configuration setting

The product page needs to be updated to use the new configuration setting. Complete the following steps:

1. In Visual Studio Code, open the **Store/Components/Pages/Products.razor** file.

2. In the **@code** section, add the following variable to store the state of the feature flag:

```
C#  
  
private decimal discountPercentage;
```

3. In the **OnInitializedAsync** method, add the following code to retrieve the value of the configuration setting:

```
C#  
  
if (saleOn) {  
    discountPercentage =  
    Convert.ToDecimal(Configuration.GetSection("eShopLite__Store__DiscountPerc  
ent").Value);  
}
```

The method should now look like the following code:

```
C#  
  
protected override async Task OnInitializedAsync()  
{  
    saleOn = await FeatureManager.IsEnabledAsync("SeasonalDiscount");  
  
    // Simulate asynchronous loading to demonstrate streaming rendering  
    products = await ProductService.GetProducts();  
  
    if (saleOn) {  
        discountPercentage =  
        Convert.ToDecimal(Configuration.GetSection("eShopLite__Store__DiscountPerc  
ent").Value);  
    }  
}
```

The above code uses the Configuration object to retrieve the value of the configuration setting. The value is stored in the `discountPercentage` variable as a decimal.

Display the discounted prices

The product page needs to be updated to display the discounted prices. Complete the following steps:

1. Replace `<td>@product.Price</td>` with this code:

C#

```
<td>
  @if (saleOn) {
    <strike>@(product.Price)</strike><br>
    @((product.Price * discountPercentage).ToString("#.##"))
  } else {
    @product.Price
  }
</td>
```

The above code checks if the seasonal sale is enabled. If it's enabled, the original price is displayed with a strike-through and the discounted price is displayed below it. If the seasonal sale isn't enabled, the original price is displayed.

Build the app

1. Ensure you've saved all your changes, and are in the **dotnet-feature-flags** directory. In the terminal, run the following command:

.NET CLI

```
dotnet publish /p:PublishProfile=DefaultContainer
```

2. Run the app using docker:

Bash

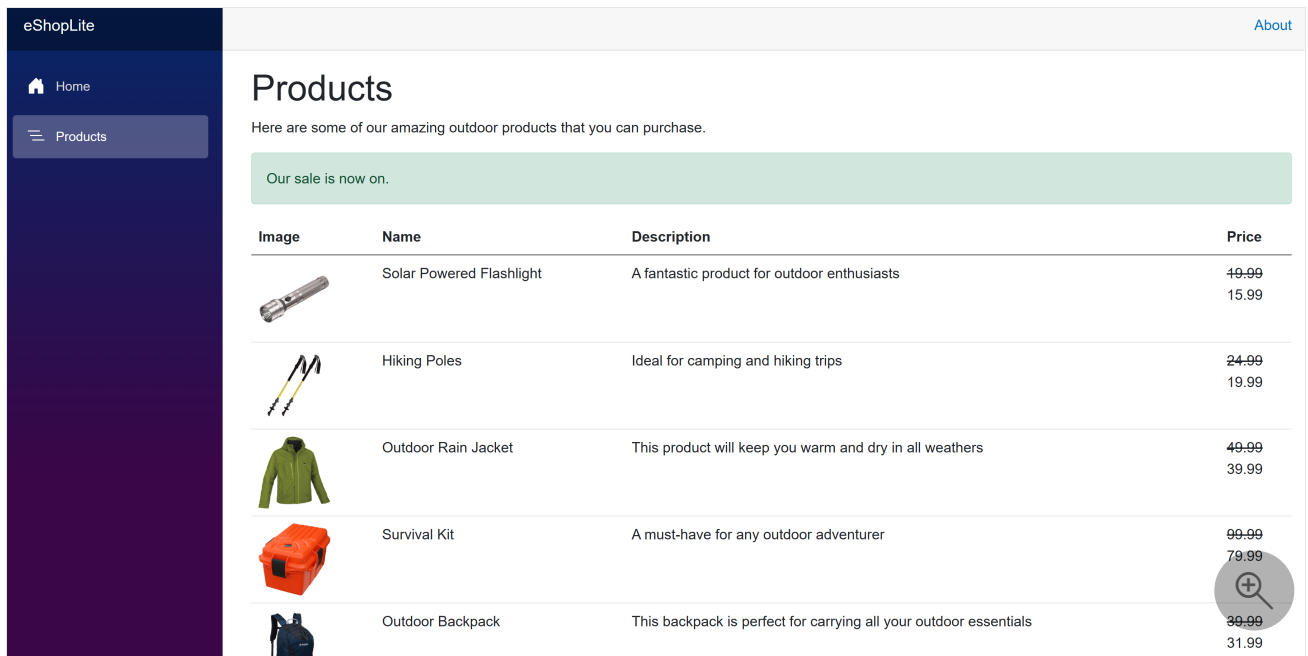
```
docker compose up
```

Test the price discount feature

To verify the feature flag works as expected in a codespace, complete the following steps:

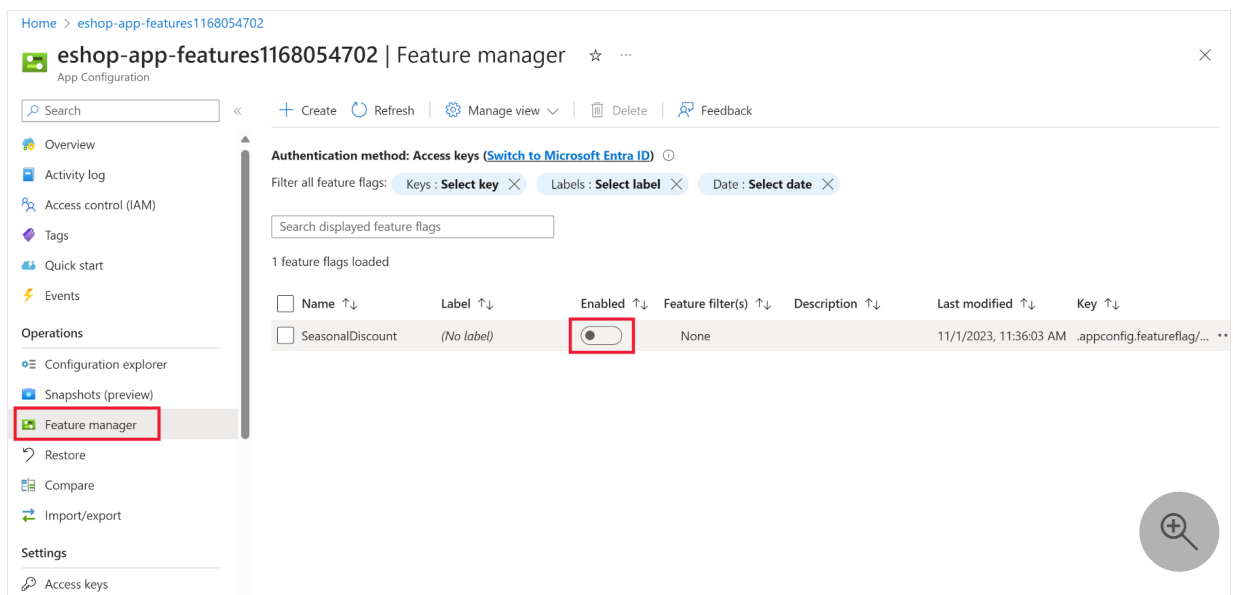
1. Switch to the **PORTS** tab, then to the right of the local address for the **Front End** port, select the globe icon. The browser opens a new tab at the homepage.
2. Select **Products**.

If you're using Visual Studio Code locally, open <http://localhost:32000/products>.



To test the feature flag is controlling the seasonal sale, complete the following steps:

1. In the Azure portal, navigate to the Azure App Configuration resource prefixed with **eshop-app-features**.
2. In the **Operations** section, select **Feature manager**.
3. Select the **SeasonalDiscount** enabled toggle to switch off this feature.



4. In your browser, return to the application.
5. Select the **Home** page, then the **Products** page.

eShopLite






Home


Products

About

Products

Here are some of our amazing outdoor products that you can purchase.

Image	Name	Description	Price
	Solar Powered Flashlight	A fantastic product for outdoor enthusiasts	19.99
	Hiking Poles	Ideal for camping and hiking trips	24.99
	Outdoor Rain Jacket	This product will keep you warm and dry in all weathers	49.99
	Survival Kit	A must-have for any outdoor adventurer	99.99
	Outdoor Backpack	This backpack is perfect for carrying all your outdoor essentials	39.99



39.99

It can take up to 30 seconds for the cache to be cleared. If the sales banner is still being shown, wait a few seconds and refresh the page again.

Next unit: Knowledge check

[Continue >](#)

Knowledge check

5 minutes

Check your knowledge

1. What's the key abstraction that supports the configuration system in ASP.NET Core apps? *

- ☐ Feature Management library
- ☐ Configuration provider
- ☐ Azure App Configuration

2. Which of the following statements is true about Azure App Configuration? *

- ☐ App Configuration is an Azure service that centrally manages app settings and feature flags.
- ☐ App Configuration doesn't encrypt your app settings at rest.
- ☐ App Configuration can't be integrated with the .NET Feature Management library.

Check your answers

Summary

2 minutes

In this module, you:

- Reviewed ASP.NET Core app configuration concepts.
- Implemented a centralized Azure App Configuration store.
- Implemented feature flags in an ASP.NET Core app.
- Implemented configuration settings in an ASP.NET Core app.

Remove Azure resources

📘 Important

It's important to deprovision the Azure resources you used in this module to avoid accruing unwanted charges.

To remove all the resources created in this module, run the following command:

Azure CLI

```
az group delete --name rg-eshop --yes
```

The preceding command deletes the resource group that contains the Azure App Configuration instance.

Cleanup Codespace

You can delete the codespace on [GitHub](#) under **By repository** where you see `MicrosoftDocs/mslearn-dotnet-cloudnative`.

Learn more about Azure App Configuration

- [Azure App Configuration documentation](#)
- [Quickstart: Add feature flags to an ASP.NET Core app](#)

- [Quickstart: Create an ASP.NET Core app with Azure App Configuration](#)
-
-