

# Implement resiliency in a cloud-native .NET microservice

1 hour

Advanced    Developer    Solution Architect    ASP.NET Core    Azure Azure Container Registry

Azure Kubernetes Service (AKS)

Learn how to make your cloud-native .NET Core microservices app fault-tolerant with minimal impact on the user.

## Learning objectives

In this module, you will:

- Understand foundational resiliency concepts.
- Observe the behavior of a microservice with no resiliency strategy.
- Implement failure handling code for HTTP requests in one microservice.
- Deploy an infrastructure-based resiliency solution to an Azure Kubernetes Service (AKS) cluster.

## Prerequisites

- Familiarity with C# and .NET development at the beginner level
- Familiarity with RESTful service concepts
- Conceptual knowledge of containers and AKS at the intermediate 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

## Application and infrastructure resiliency

5 min

## Implement application resiliency

10 min

## Exercise - Implement application resiliency

9 min

## Implement infrastructure resiliency with Kubernetes

15 min

## Exercise - Implement infrastructure resiliency with Kubernetes

11 min

## Knowledge check

7 min

## Summary

# Introduction

1 minute

Imagine that you're a software developer for an online retailer named *eShop*. The retailer uses a microservices-based architecture that's native to the cloud, and uses .NET for its online storefront. The solution includes a NET API referred to as the product service. The store service makes calls to the backend products API to get details of the products for sale.

This module focuses on *resiliency*, which is the ability of an application or service to handle problems. Resiliency helps make your app fault-tolerant in a way that has the lowest possible impact on the user. The following resilience approaches are explored:

- Using a code-based approach
- Using an infrastructure-based approach

You'll modify the app to include some code-based resiliency handling policies in a microservice. You'll also reconfigure your Azure Kubernetes Service (AKS) deployment to implement an infrastructure-based solution.

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

In this module, you will:

- Understand foundational resiliency concepts.
- Observe the behavior of a microservice that has no resiliency strategy.
- Implement failure handling code for HTTP requests in one microservice.
- Deploy an infrastructure-based resiliency solution to an AKS cluster.

## Prerequisites

- Familiarity with C# and .NET development at the beginner level.
- Familiarity with RESTful service concepts.
- Conceptual knowledge of containers and AKS at the intermediate level.
- Ability to run development containers GitHub Codespaces or in Visual Studio Code.

# Application and infrastructure resiliency

5 minutes

Resiliency is the ability to recover from transient failures. The app's recovery strategy restores normal function with minimal user impact. Failures can happen in cloud environments, and your app should respond in a way that minimizes downtime and data loss. In an ideal situation, your app handles failures gracefully without the user ever knowing there was a problem.

Because microservice environments can be volatile, design your apps to expect and handle partial failures. A partial failure, for example, can include code exceptions, network outages, unresponsive server processes, or hardware failures. Even planned activities, such as moving containers to a different node within a Kubernetes cluster, can cause a transient failure.

## Resiliency approaches

In designing resilient applications, you often have to choose between failing fast and graceful degradation. Failing fast means the application will immediately throw an error or exception when something goes wrong, rather than try to recover or work around the problem. This allows issues to be identified and fixed quickly. Graceful degradation means the application will try to keep operating in a limited capacity even when some component fails.

In cloud-native applications it's important for services to handle failures gracefully rather than fail fast. Since microservices are decentralized and independently deployable, partial failures are expected. Failing fast would allow a failure in one service to quickly take down dependent services, which reduce overall system resiliency. Instead, microservices should be coded to anticipate and tolerate both internal and external service failures. This graceful degradation allows the overall system to continue operating even if some services are disrupted. Critical user-facing functions can be sustained, avoiding a complete outage. Graceful failure also allows disturbed services time to recover or self-heal before impacting the rest of the system. So for microservices-based applications, graceful degradation better aligns with resiliency best practices like fault isolation and rapid recovery. It prevents local incidents from cascading across the system.

There are two fundamental approaches to support a graceful degradation with resiliency: application and infrastructure. Each approach has benefits and drawbacks. Both approaches

can be appropriate depending on the situation. This module explains how to implement both *code-based* and *infrastructure-based* resiliency.

## Code-based resiliency

To implement code-based resiliency, .NET has an extension library for resilience and transient failure handling, `Microsoft.Extensions.Http.Resilience`.

It uses a fluent, easy-to-understand syntax to build failure-handling code in a thread-safe manner. There are several resilience policies that define failure-handling behavior. In this module, you apply the Retry and Circuit Breaker strategies to HTTP client operations.

### Retry strategy

A *Retry* strategy is exactly what the name implies. The request is retried after a short wait if an error response is received. The wait time increases with each retry. The increase can be linear or exponential.

After the maximum number of retries is reached, the strategy gives up and throws an exception. From the user's perspective, the app usually takes longer to complete some operations. The app might also take some time before informing the user that it couldn't complete the operation.

### Circuit Breaker strategy

A *Circuit Breaker* strategy gives a target service a break after a repeated number of failures by pausing trying to communicate with it. The service could be experiencing a serious problem and be temporarily unable to respond. After a defined number of consecutive failures, the connection attempts are paused, *opening* the circuit. During this wait, additional operations on the target service fail immediately without even trying to connect the service. After the wait time has elapsed, the operation is attempted again. If the service successfully responds, the circuit is *closed* and the system goes back to normal.

## Infrastructure-based resiliency

To implement infrastructure-based resiliency, you can use a *service mesh*. Aside from resiliency without changing code, a service mesh provides traffic management, policy, security, strong identity, and observability. Your app is decoupled from these operational capabilities, which are moved to the infrastructure layer.

## Comparison to code-based approaches

An infrastructure-based resiliency approach can use a metrics-based view that allows it to adapt dynamically to cluster conditions in real time. This approach adds another dimension to managing the cluster, but doesn't add any code.

With a code-based approach you:

- Are required to guess which retry and timeout parameters are appropriate.
- Focus on a specific HTTP request.

There's no reasonable way to respond to an infrastructure failure in your app's code. Consider the hundreds or thousands of requests that are being processed simultaneously. Even a retry with exponential back-off (times request count) can flood a service.

In contrast, infrastructure-based approaches are unaware of app internals. For example, complex database transactions are invisible to service meshes. Such transactions can only be protected from failure with a code based approach.

In upcoming units, you'll implement resilience for a microservice based app using .NET HTTP resiliency in code and a Linkerd service mesh.

# Implement application resiliency

10 minutes

The resiliency features of .NET are built upon the Polly project and made available through `Microsoft.Extensions`. You can add a standard resilience strategy that uses sensible defaults by adding a single line of code to your app.

## Add resilience to your app

To add resilience to an app built using a microservices architecture, using HTTP requests between individual services, take these steps:

1. Add the `Microsoft.Extensions.Http.Resilience` package to your project.
2. Add a resilience handler to your `HttpClient` service calls.
3. Configure the resilience strategy.

## Add the NuGet package to your project

Run the following command to add the resiliency NuGet package:

.NET CLI

```
dotnet add package Microsoft.Extensions.Http.Resilience
```

Running this command from the terminal in the apps project folder will add the package reference to the project file.

In your application's startup class then add the following using statement:

C#

```
using Microsoft.Extensions.Http.Resilience;
```

## Add a resilience strategy

You can now add a standard resilience strategy to your `HttpClient` service. .NET provides this out-of-the-box configuration combining a number of strategies.





The request handler goes through each of the above strategies in order from left to right:

- **Total request timeout strategy:** This sets a total amount of time that the request can take. You can think of this as setting the upper time limit for all the other strategies.
- **Retry strategy:** This strategy controls the options on number of retries, backoff, and jitter. These options can't exceed the total timeout set in the previous strategy.
- **Circuit breaker strategy:** This strategy opens the circuit if the failure ratio exceeds the threshold.
- **Attempt timeout strategy:** This strategy sets a timeout for each individual request. If the request takes longer than this time then an exception is thrown.

You can add this standard strategy, with all the default values by adding this extension method:

C#

```
.AddStandardResilienceHandler();
```

For example if you have declared a `WebApplication`, and you want to add a resilience strategy to the `HttpClient` service use this code:

C#

```
builder.Services.AddHttpClient<ServiceBeingCalled>(httpClient =>  
{  
    httpClient.BaseAddress = new Uri("https://service.endpoint/");  
}).AddStandardResilienceHandler();
```

The first line of the above code adds a standard resilience handler to the `HttpClient`. This will use all the default settings for the retry and circuit breaker strategies.

## Configure the resilience strategy

You can change the default values of any of the strategies by specifying new options, for example:

C#

```
.AddStandardResilienceHandler( options =>  
{
```

```
options.RetryOptions.RetryCount = 10;  
options.RetryOptions.BaseDelay = TimeSpan.FromSeconds(1);  
});
```

This code changes the retry strategy defaults to have a maximum number of retries of 10, to use a linear back off, and use a base delay of 1 second.

The options you choose have to be compatible with each other. For example, if the total time remains as its default of 30 seconds, then the retry options above will cause an exception. This is an error because the exponential backoff setting would cause the total time to complete the 10 retries to be 2046 seconds. This is a runtime exception, not a compile time error.

The following table lists the options available for each of the strategies.

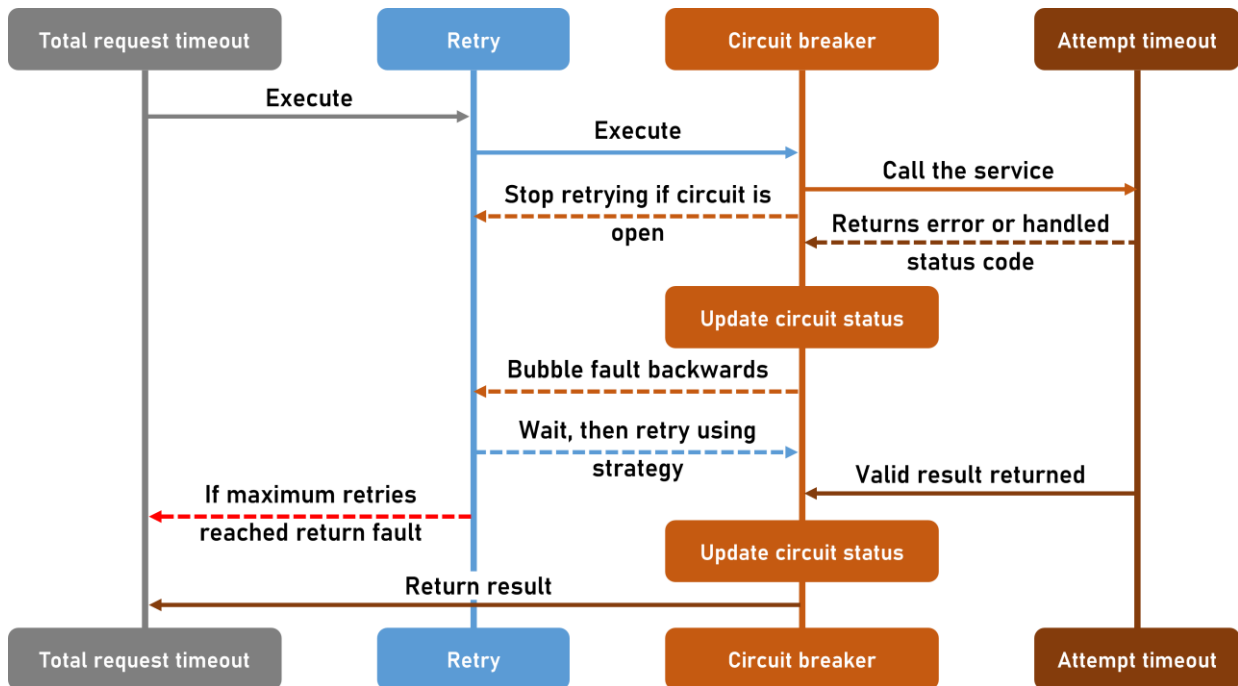
Total request timeout options	Description
TotalTimeout	The total amount of time that the request can take. The default is 30 seconds.
OnTimeout	A callback function that's invoked when the request times out. The default is null.

Retry options	Description
RetryCount	The maximum number of retries. The default is 3.
BackoffType	The type of backoff to use. You can choose between linear and exponential. The default is exponential.
UseJitter	Whether to add jitter to the backoff. Jitter adds randomness to the delay to help reduce spikes in load. The default is true.
BaseDelay	The delay between retries. The default is 2 seconds.

Circuit breaker options	Description
BreakDuration	The duration of the circuit break. The default is 5 seconds.
FailureRatio	The ratio of failed requests to successful requests that will open the circuit. The default is 0.1.
SamplingDuration	The duration of time that the failure ratio is calculated over. The default is 30 seconds.
OnClosed	A callback function that's invoked when the circuit is closed. The default is null.
OnHalfOpened	A callback function that's invoked when the circuit is half-open. The default is null.
OnOpened	A callback function that's invoked when the circuit is opened. The default is null.

 [Expand table](#)

Attempt timeout options	Description
Timeout	The amount of time that the request can take. The default is 2 seconds.
OnTimeout	A callback function that's invoked when the request times out. The default is null.



The sequence diagram above shows how each of the strategies work together in a standard resiliency strategy. To begin with the limiting factor of how long a request can take is controlled by the total timeout strategy. The retry strategy must then be set to have a maximum number of retries that will complete within the total timeout. The circuit breaker strategy will open the circuit if the failure ratio exceeds the threshold set for it. The attempt timeout strategy sets a timeout for each individual request. If the request takes longer than this time then an exception is thrown.

# Exercise - Implement application resiliency

9 minutes

The eShop project has two services that communicate with each other using HTTP requests. The `store` service calls the `product` service to get the list of all the current products available to buy.

The current version of the app has no resiliency handling. If the `product` service is unavailable, the `store` service returns an error to the customers and asks them to try again later. This behavior isn't a good user experience.

Your manager asks you to add resilience to the app, so that the `store` service retries the backend service call if it fails.

In this exercise, you add resiliency to an existing cloud-native app and test your fix.

## 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 the process completes, you see the code files for the exercise. The code to use for the remainder of this module is in the `/dotnet-resiliency` 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-resiliency` from the drop down. Visual Studio Code creates your development container locally.

# Build and run the app

1. In the bottom panel, select to the **TERMINAL** tab and run the following command go to the code root:

```
cli
```

```
cd dotnet-resiliency
```

2. Run the following command to build the eShop app images:

```
Bash
```

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

3. Once the build completes, run the following command to start the app:

```
Bash
```

```
docker compose up
```

4. In the bottom panel, select to the **PORTS** tab, then in the Forwarded Address column of the table, select the **Open in Browser** icon for the **Front End (32000)** port.

If you're running the app locally, open a browser window to view `http://localhost:32000/products`.

5. The eShop app should be running. Select the **Products** menu item, you should see the list of products.

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

# Test the current resiliency

Stop the product service to see what happens to the app.

1. Go back to your codespace, and in the **TERMINAL** tab select + to open a new bash terminal.
2. Run the following docker command to list the running containers:

Bash

```
docker ps
```

You should see the list of currently running containers, for example:

Bash

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
c08285e8aaa4	storeimage	"dotnet Store.dll"	8 minutes ago	Up 8 minutes	80/tcp, 443/tcp, 0.0.0.0:5902->8080/tcp, :::5902->8080/tcp
6ba80f3c7ab0	productservice	"dotnet Products.dll"	8 minutes ago	Up 8 minutes	80/tcp, 443/tcp, 0.0.0.0:5200->8080/tcp, :::5200->8080/tcp
cd0c822a5222	vsc-	eshoplite-958868d22c9851dd911b2423199bfc782861d1a8f7afac48e5096a1b7516082f	27 minutes ago	Up 27 minutes	

3. Look for the CONTAINER ID for the **productservice** container. In the above example, the ID is **6ba80f3c7ab0**.
4. Stop your product service with this docker command:

Bash

```
docker stop <CONTAINER ID>
```

Where the <CONTAINER ID> is the ID you found in the previous step. For example:

Bash

```
docker stop 6ba80f3c7ab0
```

5. Go back to the browser tab running the app and refresh the page. You should see an error message:

*There is a problem loading our products. Please try again later.*

6. Go back to your codespace, and in the **TERMINAL** select the **docker** terminal and press `Ctrl + C` to stop the app. You should see:

Bash

```
Gracefully stopping... (press Ctrl+C again to force)
Aborting on container exit...
[+] Stopping 2/1
  ✓ Container eshoplite-frontend-1   Stopped
  0.3s
  ✓ Container eshoplite-backend-1   Stopped
  0.0s
canceled
```

## Add resiliency to the app

The first steps to make your app more resilient are to add the

`Microsoft.Extensions.Http.Resilience` NuGet package to the project. You can then use it in `Program.cs`.

## Add the `Microsoft.Extensions.Http.Resilience` package

1. In your codespace, on the **TERMINAL** tab, navigate to the **Store** project folder:

Bash

```
cd Store
```

2. Run the following command to add the resiliency NuGet package:

Bash

```
dotnet add package Microsoft.Extensions.Http.Resilience
```

Running this command from the terminal in the apps project folder adds the package reference to the `Store.csproj` project file.

3. In the **EXPLORER** sidebar, select `Program.cs`.



4. At the top of the file, add the following using statement:

C#

```
using Microsoft.Extensions.Http.Resilience;
```

## Add a standard resilience strategy

1. At Line 13, before `;`, add this code:

C#

```
.AddStandardResilienceHandler()
```

Your code should look like this:

C#

```
builder.Services.AddHttpClient<ProductService>(c =>
{
    var url = builder.Configuration["ProductEndpoint"] ?? throw new
InvalidOperationException("ProductEndpoint is not set");

    c.BaseAddress = new(url);
}).AddStandardResilienceHandler();
```

The above code adds a standard resilience handler to the `HttpClient`. The handler uses all the default settings for the standard resilience strategy.

No other code changes are needed to your app. Let's run the app and test the resiliency.

2. Run the following commands to rebuild the eShop app:

Bash

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

3. When the build completes, run the following command to start the app:

Bash

```
docker compose up
```

4. Go back to the browser tab running the app and refresh the product page. You should see the list of products.
5. Go back to your codespace, and in the **TERMINAL** tab select the second bash terminal. Copy the CONTAINER ID for the **productservice** container.
6. Rerun the docker stop command:

Bash

```
docker stop <CONTAINER ID>
```

7. Go back to the browser tab running the app and refresh the product page. This time, it should take a little longer until you see the app's error message:

*There's a problem loading our products. Please try again later.*

Let's check the logs to see if our resilience strategy is working.

8. Go back to your codespace, and in the **TERMINAL** tab select the **docker** terminal.
9. In the terminal, press `Ctrl` + `C` to stop the app running.
10. In the log messages, scroll up until you find references to **Polly**.

Bash

```
eshoplite-frontend-1 | warn: Polly[3]  
eshoplite-frontend-1 |           Execution attempt. Source: 'ProductService-  
standard//Standard-Retry', Operation Key: '', Result: 'Name or service not  
known (backend:8080)', Handled: 'True', Attempt: '2', Execution Time:  
'27.2703'
```

You should see many messages like this; each one is a retry attempt. The above message shows the second attempt, and the time it took to execute.

## Configure a resilience strategy

When you add resiliency to your app you're balancing the need to respond quickly to your users, with the need to not overload any backend services. Only you can decide if the default options meet your businesses needs.

In this example, you'd like the store service to wait a little longer, to give the store service a chance to recover.

1. In the code window for Program.cs, change the code at line 13 to:

```
C#  
  
.AddStandardResilienceHandler(options =>  
{  
    options.Retry.MaxRetryAttempts = 7;  
});
```

The above code changes the retry strategy defaults to have a maximum number of retries to seven. Remember the strategy is an exponential backoff, so the total time is around 5 minutes.

2. Stop docker up with `Ctrl + C`. Then run the following command to rebuild the eShop app:

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

3. When the build completes, run the following command to start the app:

```
Bash  
  
docker compose up
```

Stop the backend service container in the bash terminal and refresh the eShop. Note it takes longer to see the error message. If you check the logs though, you can see that the retry strategy only retried five times. The last message from Polly is:

```
Bash  
  
Polly.Timeout.TimeoutRejectedException: The operation didn't complete  
within the allowed timeout of '00:00:30'.
```

The above message tells you that the total request timeout stops the maximum number of retries from being reached. You can fix the problem by increasing the total request timeout.

4. In the terminal, press `Ctrl + C` to stop the app.

5. In the code window for Program.cs, change the code at line 13 to:

C#

```
.AddStandardResilienceHandler(options =>
{
    options.Retry.RetryCount = 7;
    options.TotalRequestTimeout = new HttpTimeoutStrategyOptions
    {
        Timeout = TimeSpan.FromMinutes(5)
    };
});
```

The above code changes the total request timeout to 260 seconds, which is now longer than the retry strategy.

With these changes you should have enough time to run the app, stop the product service, check the terminal logs for retry attempts, refresh the eShop to see the loading message, and finally restart the product service to successfully see the list of products.

6. Run the following command to rebuild the eShop app:

Bash

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

7. When the build completes, run the following command to start the app:

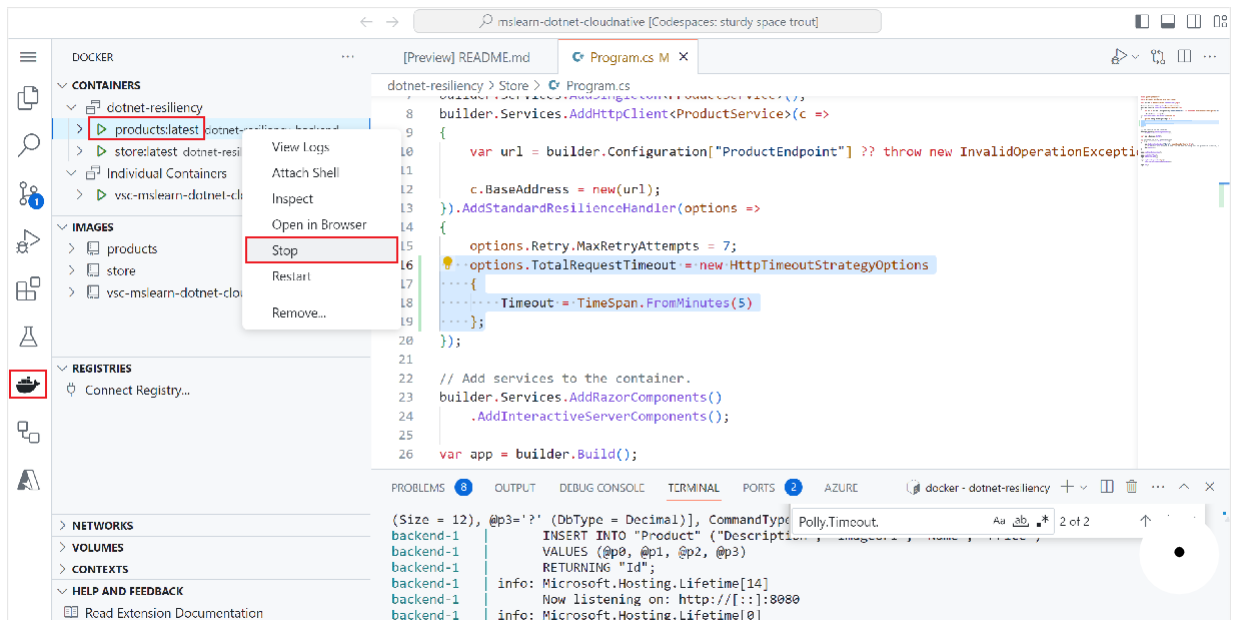
Bash

```
docker compose up
```

## Test the new resiliency options

To help test the app in your container, use the Docker extension. The extension provides a GUI to view and control the state of containers.

1. From the left menu, select the **Docker** icon.



2. In the **DOCKERS** panel, under **CONTAINERS**, right-click the **products** container and select **Stop**.
3. Go back to the browser tab running the app and refresh the product page. You should see the **Loading...** message.
4. Go back to your codespace, and in the **TERMINAL** tab, select the **docker** terminal. The resilience strategy is working.
5. In the **DOCKERS** panel, under **CONTAINERS**, right-click the **products** container and select **Start**.
6. Go back to the browser tab running the app. Wait and the app should recover showing the list the products.
7. In the Terminal, stop docker with `Ctrl + C`.

# Implement infrastructure resiliency with Kubernetes

15 minutes

To implement infrastructure-based resiliency, you can use a *service mesh*. Aside from resiliency without changing code, a service mesh provides traffic management, policy, security, strong identity, and observability. Your app is decoupled from these operational capabilities, which are moved to the infrastructure layer. Architecturally speaking, a service mesh is composed of two components: a control plane and a data plane.



The *control plane* component has many components that support managing the service mesh. The components inventory typically includes:

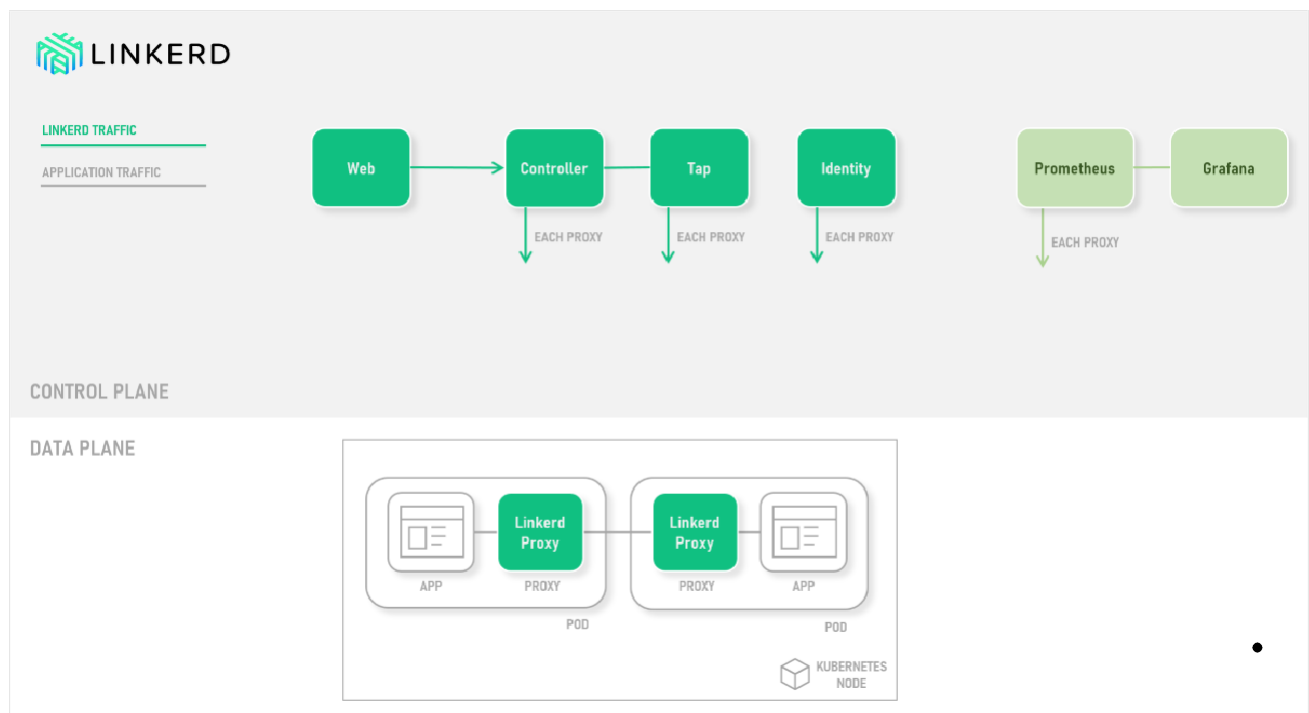
- A management interface, which could be a UI or an API.
- Rules and policy definitions that define how the service mesh should implement specific capabilities.
- Security management for things like strong identity and certificates for mTLS.

- Metrics or observability to collect and aggregate metrics and telemetry from the apps.

The *data plane* component consists of proxies that are transparently injected alongside each service; this is known as the Sidecar pattern. Each proxy is configured to control the network traffic in and out of the pod that contains your service. This configuration allows each proxy to be configured to:

- Secure traffic via mTLS.
- Dynamically route traffic.
- Apply policies to traffic.
- Collect metrics and tracing information.

Some popular service mesh options for Kubernetes clusters include Linkerd, Istio, and Consul. This module focuses on Linkerd. The following diagram shows interactions between components within the data and control planes:



## Comparison to code-based approaches

Linkerd's principal fault-handling strategy comprises [retries and timeouts](#). Because Linkerd has a systemic view of the entire cluster, it can employ resiliency strategies in novel ways. An example is retrying in such a way as to add a maximum of 20 percent additional load on the target service. Linkerd's metrics-based view allows it to adapt dynamically to cluster conditions in real time. This approach adds another dimension to managing the cluster, but doesn't add any code.

With a code-based approach, such as with Polly, you:

- Are required to guess which retry and timeout parameters are appropriate.
- Focus on a specific HTTP request.

There's no reasonable way to respond to an infrastructure failure in your app's code. Consider the hundreds or thousands of requests that are being processed simultaneously. Even a retry with exponential back-off (times request count) can flood a service.

In contrast, infrastructure-based approaches like Linkerd are unaware of app internals. For example, complex database transactions are invisible to Linkerd. Such transactions can be protected from failure with Polly.

In upcoming units, you'll implement resilience for the coupon service by using Polly and Linkerd.



# Exercise - Implement infrastructure resiliency with Kubernetes

11 minutes

In the previous unit, you implemented resiliency by adding failure-handling code using .NET native resilience extension. However, this change only applies to the service that you changed. Updating a large app with many services would be nontrivial.

Instead of using *code-based* resiliency, this unit uses an approach called *infrastructure-based* resiliency that spans the entire app. You will:

- Redeploy the app without any resiliency into Kubernetes.
- Deploy Linkerd in your Kubernetes cluster.
- Configure the app to use Linkerd for resiliency.
- Explore the app behavior with Linkerd.

## Redeploy the app

Before applying Linkerd, revert the app to a state before code-based resiliency was added. To revert, follow these steps:

1. In the bottom panel, select to the **TERMINAL** tab and run the following git commands to undo your changes:

Bash

```
cd Store
git checkout Program.cs
git checkout Store.csproj
cd ..
dotnet publish /p:PublishProfile=DefaultContainer
```

## Install Kubernetes

In your codespace, install Kubernetes and k3d. k3d is a tool that runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your local machine. It's useful for testing Kubernetes deployments locally and runs well inside a codespace.

Run these commands to install Kubernetes and MiniKube:

Bash

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg
--dearmor -o /etc/apt/kubernetes-apt-keyring.gpg

echo 'deb [signed-by=/etc/apt/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/
core:/stable:/v1.28/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update
sudo apt-get install -y kubectl

curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
k3d cluster create devcluster --config k3d.yml
```

## Deploy the eShop services to Docker Hub

The local images of your services that you build need to be hosted in a container registry to be deployable into Kubernetes. In this unit, you use Docker Hub as your container registry.

Run these commands to push your images to Docker Hub:

Bash

```
sudo docker login

sudo docker tag products [your username]/productservice
sudo docker tag store [your username]/storeimage

sudo docker push [your username]/productservice
sudo docker push [your username]/storeimage
```

## Convert your docker-compose file to Kubernetes manifests

At the moment, you define how your app runs in docker. Kubernetes uses a different format for defining how your app runs. You can use a tool called Kompose to convert your docker-compose file to Kubernetes manifests.

1. You need to edit these files to use the images you pushed to Docker Hub.
2. In the codespace, open the file *backend-deploy.yml*.
3. Change this line:

```
YAML
containers:
  - image: [YOUR DOCKER USER NAME]/productservice:latest
```

Replace the placeholder [YOUR DOCKER USER NAME] with your actual Docker username.

4. Repeat these steps for the **frontend-deploy.yml** file.

5. Change this line:

```
YAML

containers:
  - name: storefrontend
    image: [YOUR DOCKER USER NAME]/storeimage:latest
```

Replace the placeholder [YOUR DOCKER USER NAME] with your actual Docker username.

6. Deploy the eShop app into Kubernetes:

```
Bash

kubectl apply -f backend-deploy.yml,frontend-deploy.yml
```

You should see output similar to the following messages:

```
Bash

deployment.apps/productsbackend created
service/productsbackend created
deployment.apps/storefrontend created
service/storefrontend created
```

7. Check that all the services are running:

```
Bash

kubectl get pods
```

You should see output similar to the following messages:

```
Bash
```

NAME	READY	STATUS	RESTARTS	AGE
backend-66f5657758-5gnkw	1/1	Running	0	20s
frontend-5c9d8dbf5f-tp456	1/1	Running	0	20s

8. Switch to the **PORTS** tab, to view the eShop running on Kubernetes select the globe icon next to **Front End (32000)** port.

## Install linkerd

The dev container needs Linkerd CLI to be installed. Run the following command to confirm that Linkerd prerequisites are satisfied:

Bash

```
curl -sL run.linkerd.io/install | sh
export PATH=$PATH:/home/vscode/.linkerd2/bin
linkerd check --pre
```

A variation of the following output appears:

Console

```
kubernetes-api
-----
✓ can initialize the client
✓ can query the Kubernetes API

kubernetes-version
-----
✓ is running the minimum Kubernetes API version
✓ is running the minimum kubectl version

pre-kubernetes-setup
-----
✓ control plane namespace does not already exist
✓ can create non-namespaced resources
✓ can create ServiceAccounts
✓ can create Services
✓ can create Deployments
✓ can create CronJobs
✓ can create ConfigMaps
✓ can create Secrets
✓ can read Secrets
✓ can read extension-apiserver-authentication configmap
✓ no clock skew detected

pre-kubernetes-capability
```

```
-----  
✓ has NET_ADMIN capability  
✓ has NET_RAW capability  
  
linkerd-version  
-----  
✓ can determine the latest version  
✓ cli is up-to-date  
  
Status check results are ✓
```

## Deploy Linkerd to Kubernetes

First, run the following command to install the Custom Resource Definitions (CRDs):

Bash

```
linkerd install --crds | kubectl apply -f -
```

Then, run the following command:

Bash

```
linkerd install --set proxyInit.runAsRoot=true | kubectl apply -f -
```

In the preceding command:

- `linkerd install` generates a Kubernetes manifest with the necessary control plane resources.
- The generated manifest is piped to `kubectl apply`, which installs those control plane resources in the Kubernetes cluster.

The first line of the output shows that the control plane was installed in its own `linkerd` namespace. The remaining output represents the objects being created.

Console

```
namespace/linkerd created  
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-identity created  
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-identity created  
serviceaccount/linkerd-identity created  
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-controller created
```

# Validate the Linkerd deployment

Run the following command:

```
Bash
```

```
linkerd check
```

The preceding command analyzes the configurations of the Linkerd CLI and control plane. If Linkerd is configured correctly, the following output is displayed:

```
Console
```

```
kubernetes-api
-----
✓ can initialize the client
✓ can query the Kubernetes API

kubernetes-version
-----
✓ is running the minimum Kubernetes API version
✓ is running the minimum kubectl version

linkerd-existence
-----
✓ 'linkerd-config' config map exists
✓ heartbeat ServiceAccount exist
✓ control plane replica sets are ready
✓ no unschedulable pods
✓ controller pod is running
✓ can initialize the client
✓ can query the control plane API

linkerd-config
-----
✓ control plane Namespace exists
✓ control plane ClusterRoles exist
✓ control plane ClusterRoleBindings exist
✓ control plane ServiceAccounts exist
✓ control plane CustomResourceDefinitions exist
✓ control plane MutatingWebhookConfigurations exist
✓ control plane ValidatingWebhookConfigurations exist
✓ control plane PodSecurityPolicies exist

linkerd-identity
-----
✓ certificate config is valid
✓ trust anchors are using supported crypto algorithm
✓ trust anchors are within their validity period
✓ trust anchors are valid for at least 60 days
```

```
✓ issuer cert is using supported crypto algorithm
✓ issuer cert is within its validity period
✓ issuer cert is valid for at least 60 days
✓ issuer cert is issued by the trust anchor
```

```
linkerd-api
```

```
-----
```

```
✓ control plane pods are ready
✓ control plane self-check
✓ [kubernetes] control plane can talk to Kubernetes
✓ [prometheus] control plane can talk to Prometheus
✓ tap api service is running
```

```
linkerd-version
```

```
-----
```

```
✓ can determine the latest version
✓ CLI is up to date
```

```
control-plane-version
```

```
-----
```

```
✓ control plane is up to date
✓ control plane and CLI versions match
```

```
linkerd-addons
```

```
-----
```

```
✓ 'linkerd-config-addons' config map exists
```

```
linkerd-grafana
```

```
-----
```

```
✓ grafana add-on service account exists
✓ grafana add-on config map exists
✓ grafana pod is running
```

```
Status check results are ✓
```

### Tip

To see a list of Linkerd components that were installed, run this command: `kubectl -n linkerd get deploy`

## Configure the app to use Linkerd

Linkerd is deployed, but it isn't configured. The app's behavior is unchanged.

Linkerd is unaware of service internals and can't determine whether it's appropriate to retry a failed request. For example, it would be a bad idea to retry a failed HTTP POST for a payment. A *service profile* is necessary for this reason. A service profile is a custom

Kubernetes resource that defines routes for the service. It also enables per-route features, such as retries and timeouts. Linkerd only retries routes configured in the service profile manifest.

For brevity, implement Linkerd only on the aggregator and coupon services. To implement Linkerd for those two services, you will:

- Modify the eShop deployments so Linkerd creates its proxy container in the pods.
- To configure retries on the coupon service's route, add a service profile object to the cluster.

## Modify the eShop deployments

The services must be configured to use Linkerd proxy containers.

1. Add the `linkerd.io/inject: enabled` annotation to the **backend-deploy.yml** file under template metadata.

YAML

```
template:
  metadata:
    annotations:
      linkerd.io/inject: enabled
  labels:
```

2. Add the `linkerd.io/inject: enabled` annotation to the **frontend-deploy.yml** file in the same place.
3. Update the deployments in the Kubernetes cluster:

Bash

```
kubectl apply -f backend-deploy.yml,frontend-deploy.yml
```

## Apply the Linkerd service profile for the product service

The service profile manifest for the product service is:

YAML

```
apiVersion: linkerd.io/v1alpha2
kind: ServiceProfile
```



```
metadata:
  name: backend
  namespace: default
spec:
  routes:
  - condition:
      method: GET
      pathRegex: /api/Product
      name: GET /v1/products
      isRetryable: true
  retryBudget:
    retryRatio: 0.2
    minRetriesPerSecond: 10
    ttl: 120s
```

The preceding manifest is configured so:

- Any idempotent HTTP GET route matching the pattern `/api/Product` can be retried.
- Retries can add no more than an extra 20 percent to the request load, plus another 10 "free" retries per second.

Run the following command to use the service profile in the Kubernetes cluster:

Bash

```
kubectl apply -f - <<EOF
apiVersion: linkerd.io/v1alpha2
kind: ServiceProfile
metadata:
  name: backend
  namespace: default
spec:
  routes:
  - condition:
      method: GET
      pathRegex: /api/Product
      name: GET /v1/products
      isRetryable: true
  retryBudget:
    retryRatio: 0.2
    minRetriesPerSecond: 10
    ttl: 120s
EOF
```

The following output appears:

Bash

```
serviceprofile.linkerd.io/backend created
```

## Install monitoring on the service mesh

Linkerd has extensions to give you extra features. Install the viz extension and view the status of the app in Linkerd's dashboard.

1. In the terminal, run this command to install the extension:

```
Bash
```

```
linkerd viz install | kubectl apply -f -
```

2. View the dashboard with this command:

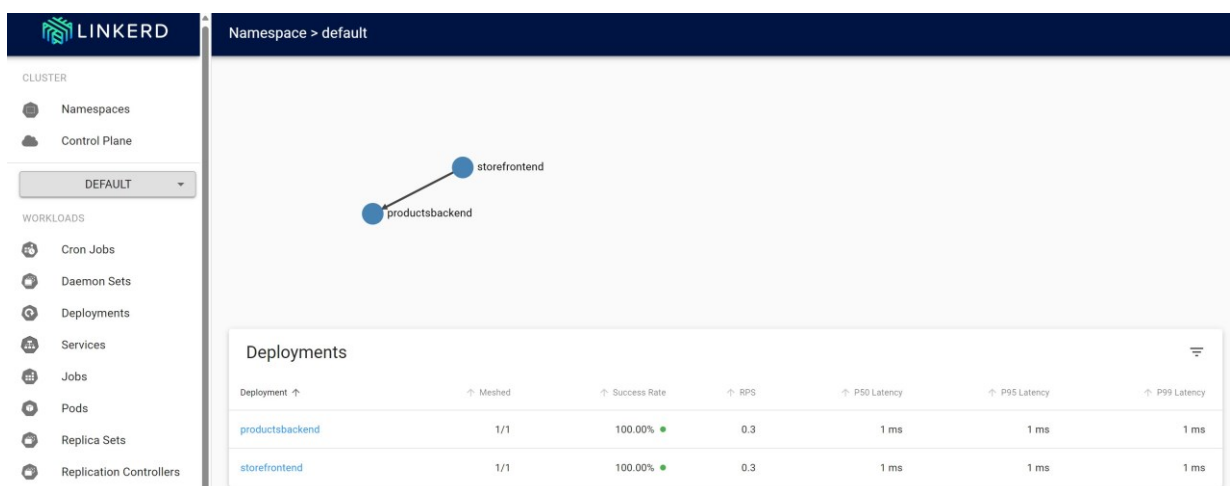
```
Bash
```

```
linkerd viz dashboard
```

Go to the **PORTS** tab and to see a new port forwarded with a process of **linkerd viz dashboard** running. Select the **Open in Browser** to open the dashboard.

3. In the Linkerd dashboard, select **Namespaces**.

4. Under HTTP Metrics, select **default**.



## Test Linkerd resiliency

After the redeployed containers are healthy, use the following steps to test the app's behavior with Linkerd:

1. Check the status of the running pods with this command:

Bash

```
kubectl get pods --all-namespaces
```

2. Stop all the product service pods:

Bash

```
kubectl scale deployment productsbackend --replicas=0
```

3. Go to the eShop web app and try to view the products. There's a delay until the error message, *"There's a problem loading our products. Please try again later."*

4. Restart the product service pods:

Bash

```
kubectl scale deployment productsbackend --replicas=1
```

5. The app should now display the products.

Linkerd follows a different approach to resiliency than what you saw with code-based resilience. Linkerd transparently retried the operation multiple times in quick succession. The app didn't need to be changed to support this behavior.

## Additional information

For more information about Linkerd configuration, see the following resources:

- [Configuring retries - Linkerd documentation](#)
- [Configuring timeouts - Linkerd documentation](#)
- [How we designed retries in Linkerd 2.2 - Linkerd blog](#)

# Knowledge check

7 minutes

## Check what you learned

1. What's the primary goal when implementing a resiliency solution for an app? \*

- ☐ Make the app error-free.
- ☐ Handling user errors transparently.
- ☐ Handling transient infrastructure failures.
- ☐ Handling major infrastructure failures.

2. Which of the following statements is correct? \*

- ☐ A code-based approach is the best option to handle failures, but it requires a significant amount of work.
- ☐ Implementing a service mesh requires little effort, but it only handles TCP connection failures.
- ☐ Both a code-based and infrastructure-based approach can have complementary roles, depending on the context.

3. Which of the following statements is true of Linkerd? \*

- ☐ It can't adjust retries and timeouts adapting to the current cluster state.
- ☐ It can't monitor the traffic between pods.
- ☐ It doesn't require any changes to the app's code.
- ☐ It can't monitor all incoming and outgoing connections.

#### 4. Which of the following statements is true about using .NET resiliency? \*

- ☐ It can't handle retries with exponential back-off.
- ☐ It can't ensure resiliency for complex database transactions.
- ☐ It does require app code changes.
- ☐ It must not be configured via code for every `HttpClient`.

## Summary

2 minutes

When you design a cloud-native application, you always need to consider **Resiliency** – the ability of an application or service to handle problems. Resiliency helps make your app fault-tolerant in a way that has the lowest possible impact on the user.

You explored the following resilience approaches in this module:

- Using a code-based approach.
- Using an infrastructure-based approach.

## Cleanup Codespace

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