.NET Microservices – Azure DevOps and AKS

Section 8: Fault Tolerance - Notes

Introduction and Fault Tolerance in Microservices

Fault tolerance refers to the ability of a system to continue functioning even when some of its components fail. In microservices architecture, individual services may fail due to various reasons, such as network issues, service unavailability, or hardware problems. Fault-tolerant systems ensure that these failures do not bring down the entire application and offer mechanisms to gracefully handle or recover from these errors.

Why is Fault Tolerance Crucial for Microservices?

Microservices are designed to be independent, small, and decentralized. While this makes the system more flexible and scalable, it also introduces additional complexities. Here's why fault tolerance is essential in microservices:

1. **Distributed Nature:** Microservices run across multiple servers, environments, and networks. This distributed architecture increases the chances of partial failure (where one or more microservices fail while others continue to function). Ensuring fault tolerance helps mitigate the impact of these failures.
2. **Service Dependencies:** Microservices often depend on each other to complete business logic. For example, an eCommerce system might have separate services for user management, order processing, payment, and inventory. If the payment service fails, the entire transaction could fail unless proper fault tolerance mechanisms are in place.
3. **Continuous Availability:** In high-availability systems, especially in cloud-based applications, downtime can have serious consequences. Customers expect services to be available 24/7, and even small outages can lead to revenue loss, poor user experience, or reputational damage.
4. **Network Latency and Timeouts:** Since microservices communicate over the network using HTTP or other protocols, network latency or timeouts can occur. Fault tolerance mechanisms help ensure that temporary issues like network delays don't crash the system.

Key Fault Tolerance Techniques in Microservices:

Here are the primary strategies to build fault tolerance into microservices:

- **Retry Mechanisms:** If a request to a microservice fails due to a temporary issue, retrying the request after a short delay can often succeed. This ensures transient errors don't cause the service to fail.
- **Exponential Backoff:** A retry mechanism where the time between retries increases exponentially to reduce load on the system and prevent further failures.

- **Circuit Breaker Pattern:** When a service is consistently failing, the circuit breaker pattern can prevent further requests to the failing service, giving it time to recover. Once the service appears to be healthy again, the circuit breaker will "close," allowing requests through.
- **Bulkhead Isolation:** This strategy isolates failures within one part of the system, preventing them from cascading and affecting other parts. This is particularly useful in microservices, where a failure in one service shouldn’t take down the entire application.
- **Fallbacks:** When a service fails, having a fallback (such as a cached response or a default value) ensures that the system can still provide some functionality instead of completely failing.

Importance of Polly in Fault Tolerance for ASP.NET Core

In ASP.NET Core, the **Polly** library is a robust and popular library for implementing fault tolerance strategies, including retries, circuit breakers, timeouts, and fallback policies. Polly allows developers to easily add fault tolerance strategies to their microservices with minimal effort.

- Polly provides a fluent API, making it easy to configure different fault-tolerance mechanisms.
- Polly integrates seamlessly with ASP.NET Core's dependency injection (DI) and HTTP clients, allowing for efficient handling of service-to-service communication failures.

Example: Microservice Scenario

Imagine an online shopping system where:

- The **PaymentService** may go down temporarily due to heavy traffic.
- The **InventoryService** might face intermittent failures due to data inconsistency.

In such cases:

- The retry mechanism (via Polly) can handle temporary PaymentService failures.
- Circuit breakers can ensure that, when InventoryService is repeatedly failing, the service is not flooded with requests, giving it time to recover.

Key Points to Remember (For Interview Preparation)

- **Fault Tolerance:** Crucial in microservices to ensure the system remains available and resilient even when individual services fail.
- **Microservices and Failure:** Service dependencies and network issues can lead to partial failures; fault tolerance helps mitigate the effects.
- **Retry Mechanisms:** Simple retry mechanisms help with transient errors.
- **Exponential Backoff and Circuit Breaker:** These advanced techniques help prevent overwhelming a failing service.

- **Polly in ASP.NET Core:** Polly is a powerful library for implementing fault tolerance patterns in .NET-based microservices.

Polly Library in ASP.NET Core

The **Polly** library is an open-source .NET library that provides a comprehensive set of fault-handling techniques like retries, circuit breakers, timeouts, fallback strategies, and bulkhead isolation. It enables you to build resilient, fault-tolerant systems by adding these policies to your ASP.NET Core applications with minimal effort.

Polly integrates smoothly with ASP.NET Core, especially for **HTTP clients** where service-to-service communication is common. It's most often used with HttpClientFactory, but can also be used with other dependencies and services.

Why Polly?

1. **Resiliency:** Adds resilience to your microservices by handling transient faults such as network issues, service timeouts, or intermittent service failures.
2. **Ease of Use:** Offers a fluent API, making it straightforward to define complex policies.
3. **Composability:** Polly allows you to combine multiple fault-handling policies (e.g., retries with circuit breakers) in a seamless manner.
4. **Flexibility:** You can use Polly for both synchronous and asynchronous operations.

Using Polly in ASP.NET Core with HttpClientFactory

Polly is commonly used with ASP.NET Core's HttpClientFactory to handle HTTP requests in a resilient way. The HttpClientFactory is a powerful feature introduced in ASP.NET Core 2.1, allowing you to create and manage instances of HttpClient in a flexible and robust manner.

Example Setup in ASP.NET Core:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder.WaitAndRetryAsync(3, retryAttempt =>
            TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
```

```
        )
    );
}
```

In the above example, Polly is used with HttpClientFactory to define a retry policy that retries a failed request up to 3 times, with an exponential backoff (2^retryAttempt seconds).

Key Concepts:

- **AddHttpClient:** Registers an HttpClient for dependency injection with a specific retry policy.
- **WaitAndRetryAsync:** Defines a retry policy that waits for an increasing amount of time between retries, improving the system's resilience to transient errors.

WaitAndRetry Policy

The **WaitAndRetry** policy in Polly is used to retry a failed operation after waiting for a specified duration. It's a basic form of fault tolerance designed for handling transient faults (temporary issues that can succeed after a short delay).

How it Works:

- Polly attempts an action (such as an HTTP request).
- If the action fails, it waits for a predefined period (e.g., 2 seconds) and then retries the action.
- This process repeats until the maximum number of retries is reached or the action succeeds.

Example with WaitAndRetry:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder.WaitAndRetryAsync(3, retryAttempt =>
            TimeSpan.FromSeconds(2))
      );
}
```

In this case, Polly will wait 2 seconds between each retry, up to a maximum of 3 retries.

Exponential Backoff

The **Exponential Backoff** is a more advanced version of the WaitAndRetry policy. Instead of waiting for a fixed duration between retries, the wait time increases exponentially with each retry attempt (e.g., 2^1 seconds, 2^2 seconds, 2^3 seconds, and so on). This helps prevent flooding a failing service with too many requests in a short period, giving it more time to recover.

Why Use Exponential Backoff?

1. **Reduces Load:** By gradually increasing the time between retries, you reduce the load on both the service and the client.
2. **Improves Recovery Chances:** If the service is under heavy load or temporarily unavailable, exponential backoff allows more time for it to recover.

Code Example for Exponential Backoff:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder.WaitAndRetryAsync(3, retryAttempt =>
            TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)) // 2^retryAttempt seconds
         )
      );
}
```

In this example:

- **1st retry:** Waits for 2 seconds (2^1)
- **2nd retry:** Waits for 4 seconds (2^2)
- **3rd retry:** Waits for 8 seconds (2^3)

Key Points to Remember (For Interview Preparation)

- **Polly Library:** Essential for implementing fault tolerance in ASP.NET Core applications. It offers various fault-handling policies such as retry, circuit breaker, timeout, and fallback.
- **WaitAndRetry Policy:** A simple retry mechanism where the system retries failed requests after waiting for a defined period.
- **Exponential Backoff:** An advanced retry strategy where the wait time between retries increases exponentially, helping prevent overwhelming a failing service.

- **Integration with HttpClientFactory:** Polly works seamlessly with HttpClientFactory in ASP.NET Core to make service-to-service communication more resilient.

Circuit Breakers

A **circuit breaker** is a fault-tolerance mechanism designed to prevent a system from repeatedly trying to execute an operation that is likely to fail. It helps to avoid overwhelming the failing service by "breaking" the circuit after a predefined number of failures, effectively blocking further attempts until the service is deemed healthy again.

Think of it as a real-life electrical circuit breaker: when too much current flows, it "trips" and prevents further current flow, protecting the system. Similarly, when too many consecutive failures occur in a service, the circuit breaker trips, preventing more requests and allowing the system to recover.

How Circuit Breaker Works:

- **Closed State:** Initially, the circuit is "closed," meaning that requests are allowed to flow through to the service.
- **Open State:** After a predefined number of consecutive failures, the circuit "opens," preventing further requests from being sent to the failing service. During this state, the system immediately fails fast for any new requests.
- **Half-Open State:** After a cool-down period, the circuit moves to a "half-open" state. During this period, a limited number of requests are allowed to pass through to test if the service has recovered.
- If the test requests succeed, the circuit is closed again. If they fail, the circuit reopens.

Code Example for Circuit Breaker with Polly:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder.CircuitBreakerAsync(
            handledEventsAllowedBeforeBreaking: 5,   // Number of exceptions before the circuit opens
            durationOfBreak: TimeSpan.FromSeconds(30) // Time the circuit remains open
         )
```

```
    );
}
```

In this example:

- After **5 consecutive failures**, the circuit opens.
- The circuit remains open for **30 seconds** before moving to the half-open state, allowing the system to check if the service has recovered.

When to Use Circuit Breakers:

- **Service Overload:** To protect an already struggling service from being overwhelmed with requests.
- **Intermittent Failures:** When dealing with services that are likely to experience short periods of downtime or failure.

Handling BrokenCircuitException

When a Polly circuit breaker is in an **open** or **half-open** state, it throws a BrokenCircuitException for any new requests, indicating that the circuit is currently preventing further execution.

Code Example of Handling BrokenCircuitException:

```
public async Task<ProductResponse> GetProductDetailsAsync(int productId)
{
  try
  {
    // Execute request through circuit breaker
    return await _httpClient.GetFromJsonAsync<ProductResponse>($"/api/products/{productId}");
  }
  catch (BrokenCircuitException ex)
  {
    // Handle the exception by returning a fallback response or logging the issue
    _logger.LogError("Circuit is currently open. Service unavailable: {Exception}", ex);
    throw new ServiceUnavailableException("Service is temporarily unavailable. Please try again later.");
  }
```

}

Here, when the circuit is open and a request is made, the BrokenCircuitException is caught and handled gracefully, preventing the application from crashing.

Key Notes:

- You can provide **fallback** behavior (e.g., returning a default response or cached data) when a BrokenCircuitException occurs.
- It's essential to log these exceptions to monitor the health of your services and circuits.

Fallback Policy

The **Fallback policy** is a crucial resiliency mechanism in Polly. It defines a default or alternative response when a particular operation fails. This is useful when you want your system to degrade gracefully instead of failing entirely.

How it Works:

- If an operation fails, Polly will invoke the fallback policy, allowing you to return a predefined result or execute alternative logic.
- This is often combined with other Polly policies like retry or circuit breaker.

Example of Fallback Policy:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder.FallbackAsync(
            fallbackAction: async (ct) =>
            {
               return new HttpResponseMessage(HttpStatusCode.OK)
               {
                  Content = new StringContent("Fallback response: service is currently unavailable")
               };
            },
```

```
            onFallbackAsync: async (exception, context) =>
            {
                _logger.LogError("Fallback executed. Exception: {Exception}", exception);
            })
        );
}
```

In this example:

- If the request fails, Polly executes the **fallbackAction** and returns a default HTTP response with a custom message.
- You can also log the fallback execution using onFallbackAsync.

Use Case for Fallback:

- **Service Unavailability:** When the primary service fails, you can return a default value or cached data to prevent full system failure.

Key Points to Remember (For Interview Preparation)

- **Circuit Breaker:** Helps prevent overwhelming a failing service by blocking further requests after consecutive failures.
    - States: **Closed**, **Open**, and **Half-Open**.
    - Use to protect services under stress and handle transient failures.
- **BrokenCircuitException:** Indicates that the circuit is open, and further execution is blocked. Handle it by providing fallback responses or alternative logic.
- **Fallback Policy:** Defines a default response or alternative execution when an operation fails, ensuring the system degrades gracefully.

Timeout Policy

The **Timeout Policy** in Polly ensures that if an operation takes longer than a specified amount of time, it is automatically abandoned and throws a TimeoutRejectedException. This is essential in microservices, where slow responses can lead to system bottlenecks or cascading failures.

By setting a timeout, you can make sure that slow or unresponsive services don't block resources indefinitely, allowing the system to handle failures more gracefully.

How Timeout Policy Works:

- You define a time limit for the execution of an operation.
- If the operation exceeds this time, Polly cancels the execution and throws an exception.
- Timeout policies are often used with retry or circuit breaker policies to handle failures robustly.

Code Example of Timeout Policy with Polly:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder
            .Or<TimeoutRejectedException>()  // Handle timeout exceptions
            .TimeoutAsync(TimeSpan.FromSeconds(5))  // Set timeout for 5 seconds
      );
}
```

In this example:

- If the HTTP call to the product service takes longer than **5 seconds**, Polly cancels the request and throws a TimeoutRejectedException.

When to Use a Timeout Policy:

- When interacting with external services that may have long response times.
- To prevent resource blocking in your microservices when certain services take too long to respond.

Best Practices for Timeout Policy:

- Combine with **retry policies** to handle transient failures effectively.
- Set reasonable timeout values based on the expected response times of your services.

Handling TimeoutRejectedException

When a timeout occurs and Polly cancels the operation, a TimeoutRejectedException is thrown. Handling this exception allows you to log the issue or provide a fallback response, preventing the entire system from failing.

Code Example of Handling TimeoutRejectedException:

```
public async Task<ProductResponse?> GetProductDetailsAsync(int productId)
{
   try
   {
      // Make an HTTP call with a timeout policy in place
      return await _httpClient.GetFromJsonAsync<ProductResponse>($"/api/products/{productId}");
   }
   catch (TimeoutRejectedException ex)
   {
      // Handle the timeout exception by logging the issue
      _logger.LogError("Request timed out. Service is unresponsive: {Exception}", ex);
      throw new ServiceUnavailableException("The service timed out. Please try again later.");
   }
}
```

In this example:

- If the request takes too long and a TimeoutRejectedException is thrown, we catch it and log the issue.
- A custom exception (ServiceUnavailableException) is thrown to notify the client that the service is currently unresponsive.

Bulkhead Isolation Policy

The **Bulkhead Isolation Policy** is a crucial mechanism in fault tolerance. It prevents a system from being overwhelmed by isolating resources for different workloads. The term "bulkhead" is borrowed from shipbuilding, where a ship's compartments (bulkheads) are isolated from one another to prevent flooding from spreading.

In microservices, bulkhead isolation ensures that a failure in one part of the system doesn't consume all resources, thereby preventing other parts of the system from being affected. This is especially useful when dealing with high-load services or when certain services might fail under pressure.

How Bulkhead Isolation Works:

- You can limit the number of concurrent executions or threads allocated to a particular resource or service.
- If the limit is reached, new requests are either queued or rejected.

Code Example of Bulkhead Isolation Policy with Polly:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddHttpClient<IProductService, ProductService>()
      .AddTransientHttpErrorPolicy(policyBuilder =>
         policyBuilder.BulkheadAsync(
            maxParallelization: 10,  // Limit to 10 concurrent requests
            maxQueuingActions: 20    // Queue up to 20 additional requests
         )
      );
}
```

In this example:

- A maximum of **10 concurrent requests** is allowed for the product service.
- If more than 10 requests are made simultaneously, up to **20 additional requests** can be queued.

If both the concurrency and queuing limits are exceeded, the system rejects new requests, ensuring other services can continue to operate without being affected.

When to Use Bulkhead Isolation:

- In services that deal with unpredictable loads or peak traffic.
- When you want to ensure that a single overloaded service does not consume all system resources, protecting other services from failure.

Key Points to Remember (For Interview Preparation)

- **Timeout Policy:** Automatically cancels an operation if it exceeds a defined time limit, preventing slow or unresponsive services from blocking resources. Use it to handle long-running operations.
    - Exception: TimeoutRejectedException is thrown when the time limit is exceeded.
- **Bulkhead Isolation Policy:** Limits the number of concurrent executions or threads for a specific service to prevent system overload. It ensures that a failure in one part of the system doesn't impact the entire system.
- **Best Practices for Combining Policies:** Use a combination of **timeout**, **retry**, **circuit breaker**, and **bulkhead** policies to build a highly resilient microservices system. Each policy addresses a different failure scenario, providing comprehensive fault tolerance

Code Explanation

This code is for implementing **fault tolerance** in an ASP.NET Core application using the **Polly** library. It handles HTTP calls to different microservices and applies various Polly policies, such as retry, circuit breaker, timeout, fallback, and bulkhead isolation, to ensure the system is resilient to transient and persistent faults.

1. Program.cs - Configuring HTTP Clients with Fault-Tolerant Policies

```
builder.Services.AddHttpClient<UsersMicroserviceClient>(client =>
{
  client.BaseAddress = new
Uri($"http://{builder.Configuration["UsersMicroserviceName"]}:{builder.Configuration["UsersMicros
ervicePort"]}");
})
.AddPolicyHandler(

builder.Services.BuildServiceProvider().GetRequiredService<IUsersMicroservicePolicies>().GetCombi
nedPolicy());
```

- AddHttpClient<UsersMicroserviceClient>: This creates a typed HttpClient for communicating with the **UsersMicroservice**.
- client.BaseAddress = new Uri(...): The BaseAddress specifies the root URL for the **UsersMicroservice** by reading the name and port from the configuration (appsettings.json or environment variables).

- AddPolicyHandler(): Adds fault tolerance using Polly policies for HTTP requests.
    - builder.Services.BuildServiceProvider().GetRequiredService<IUsersMicroservicePolicies>(): This retrieves an implementation of IUsersMicroservicePolicies, which contains the policies for fault tolerance.
    - .GetCombinedPolicy(): This method returns a **combined policy** (retry, circuit breaker, and timeout).

```
builder.Services.AddHttpClient<ProductsMicroserviceClient>(client =>
{
  client.BaseAddress = new Uri($"http://{builder.Configuration["ProductsMicroserviceName"]}:{builder.Configuration["ProductsMicroservicePort"]}");
})
.AddPolicyHandler(

builder.Services.BuildServiceProvider().GetRequiredService<IProductsMicroservicePolicies>().GetFallbackPolicy())
.AddPolicyHandler(

builder.Services.BuildServiceProvider().GetRequiredService<IProductsMicroservicePolicies>().GetBulkheadIsolationPolicy());
```

- AddHttpClient<ProductsMicroserviceClient>: Configures the HttpClient for **ProductsMicroservice**.
- client.BaseAddress = new Uri(...): The root URL is configured similarly to the UsersMicroservice.
- **Policies for ProductsMicroservice**:
    - GetFallbackPolicy(): Adds a fallback policy, where, if the service fails, dummy data will be returned instead.
    - GetBulkheadIsolationPolicy(): Adds a **Bulkhead Isolation policy** to limit concurrent requests and avoid overwhelming the service.

2. Polly Policies - Interfaces and Classes

IPollyPolicies Interface

This interface defines three core Polly policies:

```
public interface IPollyPolicies
{
  IAsyncPolicy<HttpResponseMessage> GetRetryPolicy(int retryCount);
  IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy(int
handledEventsAllowedBeforeBreaking, TimeSpan durationOfBreak);
  IAsyncPolicy<HttpResponseMessage> GetTimeoutPolicy(TimeSpan timeout);
}
```

- **GetRetryPolicy()**: Defines a retry mechanism where requests are retried a specified number of times before failing.
- **GetCircuitBreakerPolicy()**: Implements a circuit breaker pattern to temporarily block requests after repeated failures.
- **GetTimeoutPolicy()**: Cancels requests that exceed a specified time limit.

IProductsMicroservicePolicies Interface

This interface defines additional policies specific to the **ProductsMicroservice**:

```
public interface IProductsMicroservicePolicies
{
  IAsyncPolicy<HttpResponseMessage> GetFallbackPolicy();
  IAsyncPolicy<HttpResponseMessage> GetBulkheadIsolationPolicy();
}
```

- **GetFallbackPolicy()**: Returns a fallback response when the request fails.
- **GetBulkheadIsolationPolicy()**: Limits the number of concurrent requests to avoid overloading the service.

IUsersMicroservicePolicies Interface

This interface defines the combined fault-tolerant policy for the **UsersMicroservice**:

```
public interface IUsersMicroservicePolicies
{
  IAsyncPolicy<HttpResponseMessage> GetCombinedPolicy();
}
```

3. Polly Policies Implementation

PollyPolicies Class - Core Retry, Circuit Breaker, Timeout Policies

```
public class PollyPolicies : IPollyPolicies
{
  private readonly ILogger<UsersMicroservicePolicies> _logger;

  public PollyPolicies(ILogger<UsersMicroservicePolicies> logger)
  {
    _logger = logger;
  }
```

- ILogger<UsersMicroservicePolicies>: Used for logging retry attempts, circuit breaker state changes, etc.

GetRetryPolicy() - Retry with Exponential Backoff

```
public IAsyncPolicy<HttpResponseMessage> GetRetryPolicy(int retryCount)
{
  AsyncRetryPolicy<HttpResponseMessage> policy = Policy.HandleResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
  .WaitAndRetryAsync(
    retryCount: retryCount,
    sleepDurationProvider: retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
    onRetry: (outcome, timespan, retryAttempt, context) =>
    {
```

```
    _logger.LogInformation($"Retry {retryAttempt} after {timespan.TotalSeconds} seconds");
  });


 return policy;
}
```

- **WaitAndRetryAsync()**: Defines the retry logic.
  - retryCount: How many times the request will be retried.
  - sleepDurationProvider: The delay between retries increases exponentially (2^retryAttempt).
  - onRetry: Logs retry attempts.

GetCircuitBreakerPolicy() - Circuit Breaker Pattern

```
public IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy(int handledEventsAllowedBeforeBreaking, TimeSpan durationOfBreak)
{
 AsyncCircuitBreakerPolicy<HttpResponseMessage> policy = Policy.HandleResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
 .CircuitBreakerAsync(
  handledEventsAllowedBeforeBreaking: handledEventsAllowedBeforeBreaking,
  durationOfBreak: durationOfBreak,
  onBreak: (outcome, timespan) =>
  {
   _logger.LogInformation($"Circuit breaker opened for {timespan.TotalMinutes} minutes due to consecutive 3 failures.");
  },
  onReset: () => {
   _logger.LogInformation($"Circuit breaker closed.");
  });


 return policy;
}
```

- **CircuitBreakerAsync()**: Blocks requests for a specified period if too many failures occur.
    - handledEventsAllowedBeforeBreaking: Number of failures allowed before the circuit "opens".
    - durationOfBreak: Time period the circuit remains open before retrying requests.

GetTimeoutPolicy() - Timeout Handling

```
public IAsyncPolicy<HttpResponseMessage> GetTimeoutPolicy(TimeSpan timeout)
{
  AsyncTimeoutPolicy<HttpResponseMessage> policy =
Policy.TimeoutAsync<HttpResponseMessage>(timeout);
  return policy;
}
```

- **TimeoutAsync()**: Cancels requests that exceed the specified timeout duration.

ProductsMicroservicePolicies Class - Fallback and Bulkhead Isolation

GetBulkheadIsolationPolicy() - Bulkhead Isolation

```
public IAsyncPolicy<HttpResponseMessage> GetBulkheadIsolationPolicy()
{
  AsyncBulkheadPolicy<HttpResponseMessage> policy =
Policy.BulkheadAsync<HttpResponseMessage>(
    maxParallelization: 2,
    maxQueuingActions: 40,
    onBulkheadRejectedAsync: (context) =>
    {
      _logger.LogWarning("BulkheadIsolation triggered. Can't send any more requests.");
      throw new BulkheadRejectedException("Bulkhead queue is full");
    });

  return policy;
}
```

- **BulkheadAsync()**: Limits the number of concurrent requests.
    - maxParallelization: Maximum concurrent requests allowed.
    - maxQueuingActions: Number of additional requests allowed in the queue.

GetFallbackPolicy() - Fallback with Dummy Data

```
public IAsyncPolicy<HttpResponseMessage> GetFallbackPolicy()
{
  AsyncFallbackPolicy<HttpResponseMessage> policy = Policy.HandleResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
  .FallbackAsync(async (context) =>
  {
    _logger.LogWarning("Fallback triggered: Returning dummy data.");
    ProductDTO product = new ProductDTO(Guid.Empty, "Temporarily Unavailable", "Temporarily Unavailable", 0, 0);
    var response = new HttpResponseMessage(HttpStatusCode.OK)
    {
      Content = new StringContent(JsonSerializer.Serialize(product), Encoding.UTF8, "application/json")
    };

    return response;
  });

  return policy;
}
```

- **FallbackAsync()**: Returns dummy data if the request fails.

UsersMicroservicePolicies Class - Combined Policies

```
public IAsyncPolicy<HttpResponseMessage> GetCombinedPolicy()
{
  var retryPolicy = _pollyPolicies.GetRetryPolicy(5);
  var circuitBreakerPolicy = _pollyPolicies.GetCircuitBreakerPolicy(3, TimeSpan.FromMinutes(2));
  var timeoutPolicy = _pollyPolicies.GetTimeoutPolicy(TimeSpan.FromSeconds(5));

  AsyncPolicyWrap<HttpResponseMessage> wrappedPolicy = Policy.WrapAsync(retryPolicy,
circuitBreakerPolicy, timeoutPolicy);
  return wrappedPolicy;
}
```

- **Policy.WrapAsync()**: Combines multiple policies (retry, circuit breaker, and timeout) into a single, fault-tolerant policy for UsersMicroservice.

Summary (For Interview Preparation)

- **Retry Policy**: Retries failed requests with exponential backoff to handle transient faults.
- **Circuit Breaker**: Blocks requests temporarily after multiple consecutive failures to prevent overwhelming a failing service.
- **Timeout Policy**: Cancels requests if they take too long to prevent resource exhaustion.
- **Fallback Policy**: Returns predefined dummy data when the service fails, ensuring a graceful failure.
- **Bulkhead Isolation**: Limits the number of concurrent and queued requests to prevent overloading the service.