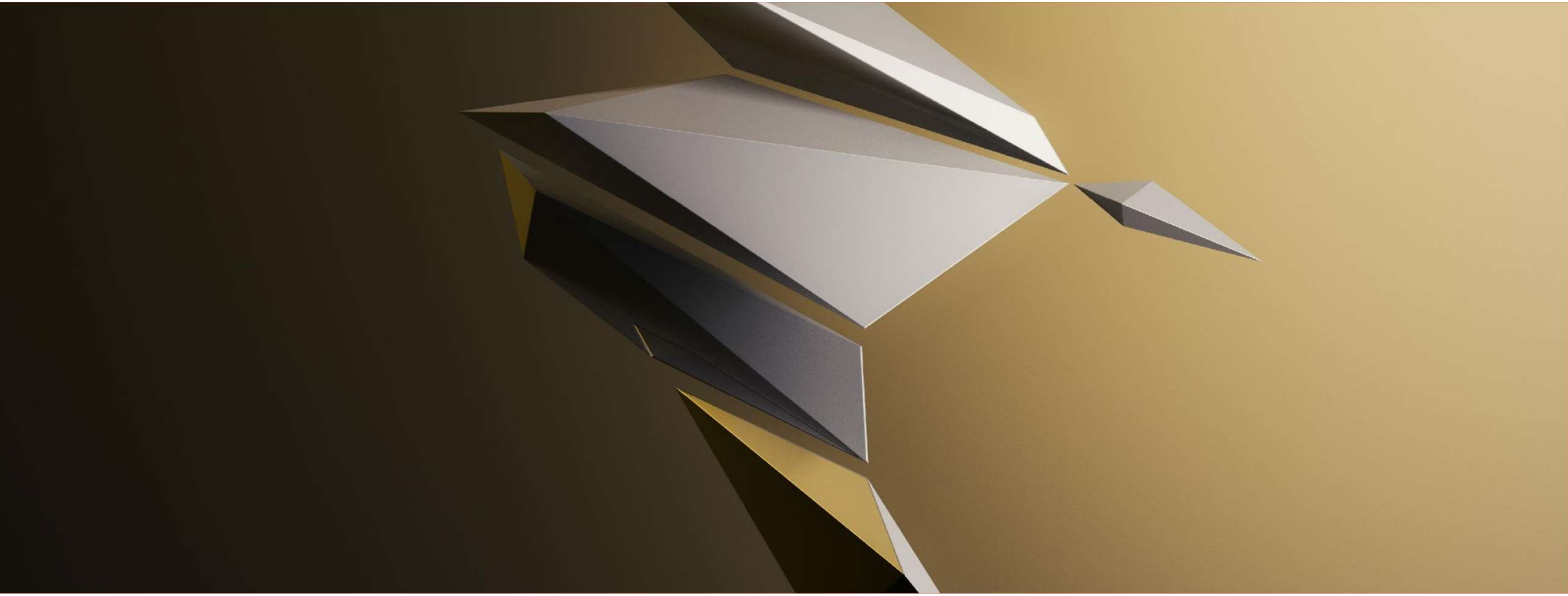
Code.Hub

The first Hub for Developers

Asp.NET Core – Microservices

Designing for Resilience



Designing for Resilience

Code.Hub

- Implementing Resilient Applications
- Handling partial failure
- Implementing retries with exponential backoff
- Implementing resilient SQL connections
- Use IHttpClientFactory for resilient HTTP requests
- HTTP call retries with exponential backoff with Polly
- Implement the Circuit Breaker pattern
- Health monitoring

Overview

Resilience - Introduction

The ability of a system or an application to withstand and recover from failures, errors, or unexpected events without compromising its overall functionality and performance.

A resilient system is designed to

1. handle and adapt to changing conditions, including partial failures or disruptions in external services,
2. maintaining essential operations and
3. ensure a reasonable level of service quality.

Audience Question.
“Why does resilience matter in a microservices-based app?”

Code.Hub

Resilience - Introduction

Resilience is a critical aspect of modern software architecture, particularly in microservices.

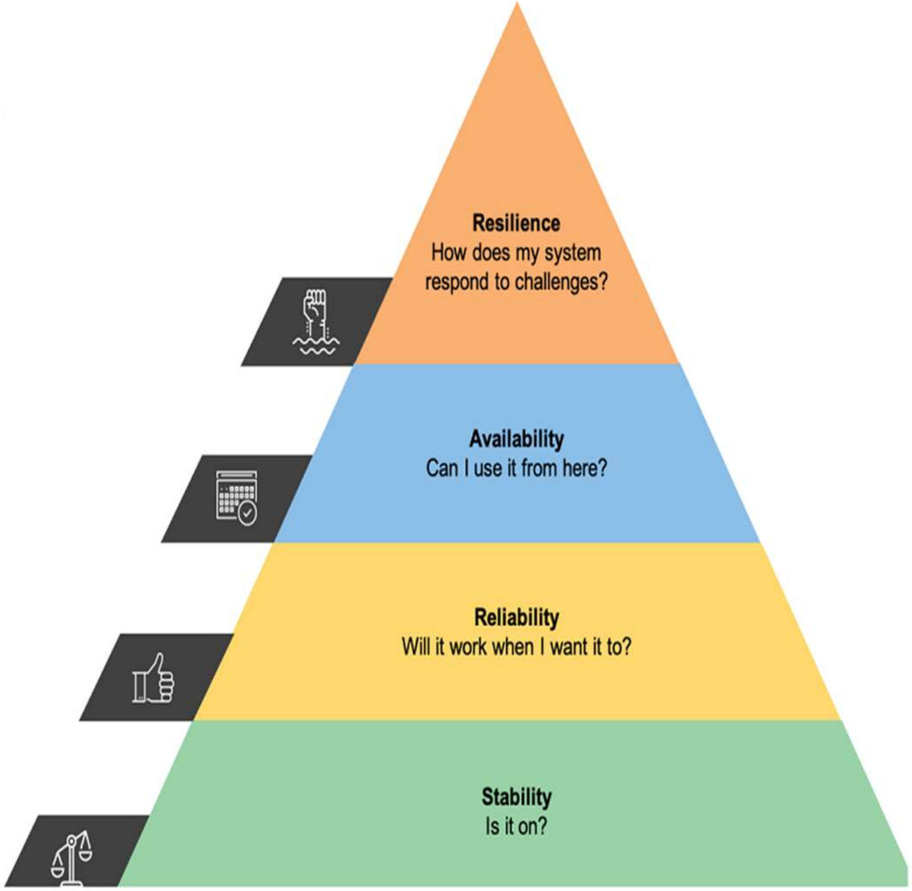
Resilience:

1) Ensures that Failures in one component should not lead to cascading failures across the entire system.
2) Involves implementing strategies, patterns, and mechanisms that enable the system to
   - recover gracefully
   - restore stability
   - continue providing valuable services even under adverse circumstances.

Designing for Resilience

Designing our apps fo Resilience involves:

- **Identifying** potentia failure points in the system
- **Planning** for gracef handling of those failures.
- **Using fault isolation** technique to prevent cascadin failures.



Code.Hub

Implementing Resilient Applications

Implementing Resilient Applications involves:

- Utilizing resilient design patterns and techniques to build applications that can withstand failures.
- Implementing fallback mechanisms to provide alternative functionality when primary services fail.

Handling Partial Failure

Handling Partial Failure means to:

- Build applications that can **tolerate partial failures,** where some components are unavailable.
- Implement strategies to ensure that the core **functionalities are maintained despite** partial failure.

Implementing Retries with Exponential Backoff

Implementing Retries with Exponential Backoff involves:

- Applying retry mechanisms with increasing time intervals between retries to reduce the load on failed services.
- Using the following Concept: Avoid overwhelming the system with retries during peak failure times.

Implementing Retries with Exponential Backoff – How to

Implementing Retries with Exponential Backoff involves:

1. **Add Required Packages**: Install the **Polly** NuGet package, which provides a convenient way to implement retry policies.
2. **Create a Retry Policy**: Define a retry policy using Polly's **Policy** class, specifying the number of retry attempts and the delay between retries.
3. **Implement the Retry Logic**: Wrap the code block or method that might encounter transient failures within the retry policy.
4. **Execute the Retry Policy**: Call the method containing the retry policy to execute the code with retries.

Implementing Retries with Exponential Backoff – Code

```
using Polly;
// Step 1: Add Required Packages - Install-Package Polly
// Step 2: Create a Retry Policy
var retryPolicy = Policy
    .Handle<YourTransientException>()
    .WaitAndRetryAsync(
        retryCount: 3, // Number of retry attempts (adjust as needed)
        sleepDurationProvider: attempt =>
TimeSpan.FromSeconds(Math.Pow(2, attempt)),
// Exponential backoff formula
        onRetry: (exception, retryCount, context) =>
        {
            // You can add custom logic here, like logging or reporting retries.
        }
    );
```

```
// Step 3: Implement the Retry Logic
public async Task DoSomethingAsync()
{
    await retryPolicy.ExecuteAsync(async () =>
    {
        // Your code block that might encounter transient failures goes here.
        // For example, calling an external API or accessing a remote
service.
        // If the operation throws 'YourTransientException', the retry policy
will handle it.
        // If the operation succeeds at any retry attempt, the policy will stop
retrying.
    });
}

// Step 4: Execute the Retry Policy
await DoSomethingAsync();
```

Implementing Resilient SQL Connections

Implementing Resilient SQL Connections is a technique that consists of

- Using connection pooling to efficiently manage connections and handle temporary connection failures.
- Configuring retry policies to handle transient SQL connection issues gracefully.

Implementing Resilient SQL Connections – How to

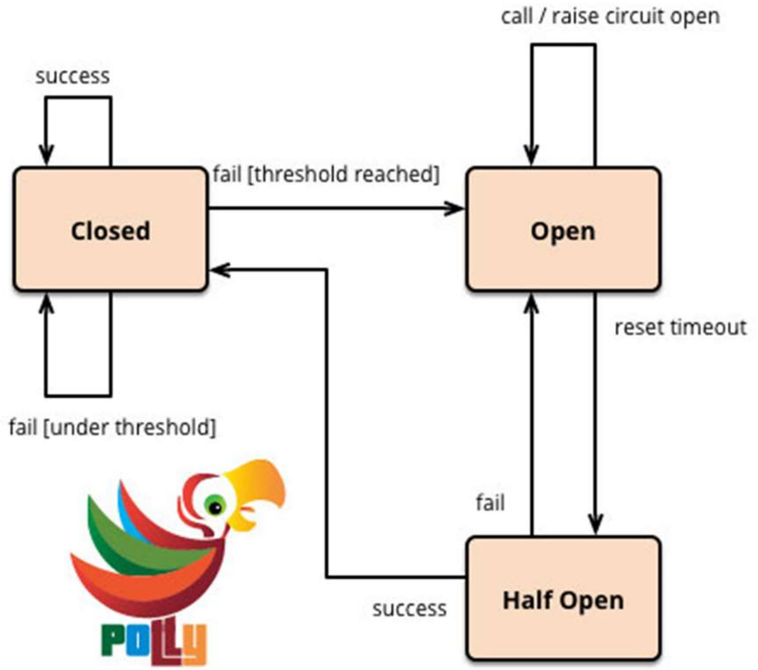
Implementing resilient SQL connections in .NET involves the following steps:

1. **Use Connection Pooling:** By default, .NET Core already uses connection pooling for SQL connections, so there's no specific configuration required. Connection pooling helps efficiently manage connections and minimizes the overhead of creating new connections.
2. **Handle Transient Failures**: Wrap your SQL operations within a retry policy to handle transient failures caused by network issues, server unavailability, or other temporary problems.
3. **Choose a Retry Library:** Use a library like Polly to implement the retry policy easily. Polly allows you to define policies for handling retries and transient faults.
4. **Configure the Retry Policy**: Define the retry policy with appropriate retry conditions, number of retries, and backoff strategies (like exponential backoff) to introduce delays between retries.
5. **Execute SQL Operations with Retry:** Wrap your SQL operations within the retry policy, so that in case of transient failures, the policy will automatically retry the SQL command.

Using IHttpClientFactory for Resilient HTTP Requests

Using IHttpClientFactory for Resilient HTTP Requests

- Leverage **IHttpClientFactory** to manage HttpClient instances efficiently.
- Implement resilient HTTP requests with **policies like retry, circuit breaker, and timeout**



Using IHttpClientFactory for Resilient HTTP Requests – How to

Using IHttpClientFactory for resilient HTTP requests in .NET involves the following steps:

1. **Add Required Packages:** Ensure you have the necessary packages installed. This typically includes the **Microsoft.Extensions.Http NuGet** package, which provides the IHttpClientFactory.
2. **Register the IHttpClientFactory:** In the ConfigureServices method of your Startup.cs class, register the **IHttpClientFactory** in the dependency injection container.
3. **Configure Resilience Policies:** Define and configure the resilience policies (e.g., retry, circuit breaker) using Polly in the **ConfigureServices** method. Optionally, you can set up policies using the **HttpClientFactory** extensions for Polly.
4. **Use IHttpClientFactory to Create an HttpClient:** In your service or controller that requires an HttpClient, inject IHttpClientFactory. Use it to create instances of HttpClient with the configured resilience policies.

HTTP Call Retries with Exponential Backoff using Polly

- Use the Polly library to easily implement HTTP call retries with exponential backoff.
- Adjust retry intervals and conditions based on the specific service's behavior.

HTTP Call Retries with Exponential Backoff using Polly

HTTP call retries with exponential backoff using Polly in .NET involves the following steps:

- **Add Required Packages:** Ensure you have the necessary packages installed, including the **Polly NuGet package.**
- **Configure Polly Policies:** Define and configure the retry policy with exponential backoff using Polly in the **ConfigureServices method of your Startup.cs class.**
- **Use Polly for HTTP Requests:** In your service or controller that makes HTTP requests, **use Polly to execute the HTTP calls** with the configured retry policy.

Code.Hub

Implementing the Circuit Breaker Pattern

Implementing the Circuit Breaker Pattern

- Employ the circuit breaker pattern **to prevent** continuous calls to a failing service.
- **Trip the circuit and allow time** for the service to recover before attempting further requests.

Implementing the Circuit Breaker Pattern – How to

Implementing the Circuit Breaker pattern in .NET involves the following steps:

- **Add Required Packages**: Ensure you have the necessary packages installed, including the Polly NuGet package.
- **Configure Polly Circuit Breaker Policy**: Define and configure the circuit breaker policy using Polly in the ConfigureServices method of your Startup.cs class.
- **Use Polly for Circuit Breaker:** In your service or controller that calls external services, use Polly to wrap the calls with the configured circuit breaker policy.

Health Monitoring

Health Monitoring techniques in microservices include:

- Implementing health checks to regularly assess the status of microservices.
- Using a health monitoring system to quickly detect and respond to issues.

Health Monitoring – What to monitor

When monitoring your app for health you must

- **Set up metrics and alerts** based on health check results to notify the operations team about potential issues promptly.
- **Analyze historical health check data** to identify patterns and address recurring failure scenarios.

Consider using one of the following apps: Grafana, Prometheus, Datadog for your monitoring.

Code.Hub

Thank you!

Code.Hub