The network is reliable isn’t it

Introduction to Polly

Andrew Clymer
andy@rocksolidknowledge.com
Agenda

• The need for resilience

• Implementing resilience with Poly
  • Retry
  • Guaranteed Response
  • Fail early
    – Timeout
    – Circuit Breaker
  • Caching
  • Isolation
    – Bulkhead
Adding reliability by hand

• Try/catch and retry
  • How many times?
  • Retry straight away, or delay?
• Resilience Framework
• Nuget package
• Fluent API for defining policy to
  • React and Handle failure
  • Reduce the risk of failure
Two flavors of reliability

• **Reactive, react to issues**
  - Retry
  - Wait and Retry
  - Circuit Breaker
  - Fallback

• **Proactive, prevent total system failure**
  - Timeout
  - Caching
  - Bulkhead Isolation
• Polly uses Policies to define strategies for dealing with or preventing failure
• Policies are thread safe, can be used again and again
• Policies can be wrapped
  • Retry 3 times, if all fails activate circuit breaker
Simple Policy

- Policy or Policy<T> as the starting point
- Handle used to identify scenario to re-act to

```
Policy<string> divideByZeroPolicy = 
    Policy<string> 
        .Handle<DivideByZeroException>(() 
            .Fallback("infinity and beyond");

string result = divideByZeroPolicy.Execute(() => (50/10).ToString());
Console.WriteLine(result);

result = divideByZeroPolicy.Execute(() => (10/0).ToString());
Console.WriteLine(result);

> 5
> infinity and beyond
```
Handle results not just exceptions

- Failure scenario can be identified by response
  - Https status codes
- By response and exceptions
- By exception type and exception details

```csharp
Policy<HttpResponseMessage>
    .HandleResult(r => !r.IsSuccessStatusCode)
    .RetryForeverAsync();

Policy<HttpResponseMessage>
    .Handle<HttpRequestException>(()
    .OrResult(r => !r.IsSuccessStatusCode)
    .RetryForeverAsync();

Policy<DbDataReader>
    .Handle<SqlException>(e => e.Number == 21)
    .RetryForever();
```
Wrapped Policy

- Policy **wrapped** around each other
- Evaluated inside out

```csharp
string lhs = "10";
string rhs = "0";

Policy<string> divideByZeroPolicy = Policy<string>
    .Handle<DivideByZeroException>()
    .Fallback("infinity and beyond");

Policy<string> nanPolicy = Policy<string>
    .Handle<FormatException>()
    .Fallback("NaN");

Policy<string> mathPolicy = nanPolicy.Wrap(divideByZeroPolicy);

string result = mathPolicy.Execute(() => (int.Parse(lhs) / int.Parse(rhs)).ToString());

Console.WriteLine(result);
```
(Http Client, Example)

- Retries the same request 3 times
- Probably too aggressive

```csharp
var retryPolicy = 
    Policy.Handle<HttpRequestException>()
    .OrResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
    .RetryAsync(3);

HttpResponseMessage response = await retryPolicy
    .ExecuteAsync(() => client.GetAsync("motd"));
```
Repair and Retry

- Retrying immediately not likely to work
- Provide **logic** to run prior to retrying
  - Back off before re-trying
  - Repair

```csharp
var retryPolicy =
    Policy.Handle<HttpRequestException>()
    .OrResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
    .RetryAsync(3 ,
                (result, retries) => {
                    if (result.Result.StatusCode == 403 ) Authorize()
                } );

HttpResponseMessage response = await retryPolicy
    .ExecuteAsync() => client.GetAsync("motd");
```
Retries the same request 3 times, with delay between each retry

```csharp
var retryPolicy =
    Policy.Handle<HttpRequestException>()
    .OrResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
    .WaitAndRetryAsync(new TimeSpan[] {
        TimeSpan.FromSeconds(1),
        TimeSpan.FromSeconds(3),
        TimeSpan.FromSeconds(5)
    });

HttpResponseMessage response = await retryPolicy.ExecuteAsync(() =>
    client.GetAsync("motd"));
```
• Policies can be applied by HttpClientFactory
  • Microsoft.Extensions.Http.Polly nuget package
• Preserves original programming model

```csharp
var retryPolicy =
    Policy<HttpResponseMessage>.Handle<HttpRequestException>()
    .OrResult<HttpResponseMessage>(r => !r.IsSuccessStatusCode)
    .WaitAndRetryAsync(...);

services.AddHttpClient("motdClient", client => {
    client.BaseAddress = new Uri("http://localhost:8081/");
})
    .AddPolicyHandler(retryPolicy);

HttpClient client = httpFactory.Create("motdClient");

HttpResponseMessage response = await client.GetAsync("motd");
```
Timeout

• Fail fast
  • Reduces the number of pending requests
  • No point waiting if you know very unlikely to work

• Two forms
  • Optimistic – relies on CancellationToken
  • Pessimistic – Spins off work async, and cancels

```csharp
var timeoutPolicy = Policy
  .TimeoutAsync<HttpResponseMessage>(TimeSpan.FromSeconds(2));
```
Circuit Breaker

• Prevents re-trying things that keep failing
• Prevents a service on restart being overloaded by a long queue of pending requests
• When broken, fails early
• Periodically re-tests, and if fixed continues to execute
Circuit Breaker state machine

- All Requests only executed when closed
- In half open state, a single request is executed, and if successful closes the circuit
Circuit Breaker Policy

- **Number of times to fail** before triggering break
- **Period** before opening partially to try one request

```csharp
var circuitBreaker = Policy<HttpResponseMessage>
    .Handle<HttpRequestException>()
    .Or<TimeoutRejectedException>()
    .OrResult(r => !r.IsSuccessStatusCode)
    .CircuitBreakerAsync(4, TimeSpan.FromSeconds(20));
```
Advanced Circuit breaker

• Reacts on *proportion* of failures

• Measures that proportion over a rolling duration, so that older failures can be excluded and have no effect

• Imposes a minimum throughput before acting, such that the circuit reacts only when statistically significant, and does not trip in 'slow' periods
Don’t react pro-actively prevent

• Prevent likely hood of failure
• Reduce load
  • Caching
• Prevent one part of the solution taking down the entire app
  • Bulk head isolation
Caching

- Reducing load on an external system reduces risk of failure
- Nuget packages for cache provider
  - Polly.Caching.Memory
  - Polly.Caching.Distributed

```csharp
IAasyncCacheProvider cache = . . .;

var cachePolicy =
    Policy.CacheAsync<List<string>>(caching, TimeSpan.FromSeconds(60));
```
A bulkhead is a wall within a ship which separates one compartment from another.

Damage to one compartment does not cause the whole ship to sink.

Increase load for one type of requests should not impact other requests.
Bulk Head Isolation Policy

- Bulk Head parts
  - Throttling
    - Maximum number of executions at any one time
  - Queue
    - Maximum number of executions waiting to be executed

```csharp
var policy = Policy.BulkheadAsync<double>(
    maxParallelization: 4,
    maxQueuingActions: 100);
```
• When things get too much, **can allocate more resource**

```csharp
var policy = Policy.BulkheadAsync<double>(
maxParallelization: 4,
maxQueuingActions: 100, _ =>
{
    // Find some more resource
});
```
Summary

• Polly delivers the building blocks to add resilience
• Building in resilience allows us to sleep