App-vNext / **Polly** Public

Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner. From version 6.0.1, Polly targets .NET Standard 1.1 and 2.0+.

www.thepollyproject.org

View license

9.7k stars 920 forks

Star Notifications

Code Issues 68 Pull requests 7 Actions Projects Wiki Secu

master Go to file

SimonCropp simplify issue templates ... on Jun 30 977

View code

README.md

Polly

Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Polly targets .NET Standard 1.1 (coverage: .NET Core 1.0, Mono, Xamarin, UWP, WP8.1+) and .NET Standard 2.0+ (coverage: .NET Core 2.0+, .NET Core 3.0, and later Mono, Xamarin and UWP targets). The nuget package also includes direct targets for .NET Framework 4.6.1 and 4.7.2.

For versions supporting earlier targets such as .NET4.0 and .NET3.5, see the supported targets grid.

We are a member of the .NET Foundation!

Keep up to date with new feature announcements, tips & tricks, and other news through www.thepollyproject.org

Installing via NuGet

```
Install-Package Polly
```

Resilience policies

Polly offers multiple resilience policies:

Policy	Premise	Aka	How does the policy mitigate?
Retry (policy family) (quickstart ; deep)	Many faults are transient and may self-correct after a short delay.	"Maybe it's just a blip"	Allows configuring automatic retries.
Circuit-breaker (policy family) (quickstart ; deep)	When a system is seriously struggling, failing fast is better than making users/callers wait. Protecting a faulting system from overload can help it recover.	"Stop doing it if it hurts" "Give that system a break"	Breaks the circuit (blocks executions) for a period, when faults exceed some pre-configured threshold.

Policy	Premise	Aka	How does the policy mitigate?
Timeout (quickstart ; deep)	Beyond a certain wait, a success result is unlikely.	"Don't wait forever"	Guarantees the caller won't have to wait beyond the timeout.
Bulkhead Isolation (quickstart ; deep)	When a process faults, multiple failing calls backing up can easily swamp resource (eg threads/CPU) in a host. A faulting downstream system can also cause 'backed-up' failing calls upstream. Both risk a faulting process bringing down a wider system.	"One fault shouldn't sink the whole ship"	Constrains the governed actions to a fixed-size resource pool, isolating their potential to affect others.
Cache (quickstart ; deep)	Some proportion of requests may be similar.	"You've asked that one before"	Provides a response from cache if known. Stores responses automatically in cache, when first retrieved.
Fallback (quickstart ; deep)	Things will still fail - plan what you will do when that happens.	"Degrade gracefully"	Defines an alternative value to be returned (or action to be executed) on failure.
PolicyWrap (quickstart ; deep)	Different faults require different strategies; resilience means using a combination.	"Defence in depth"	Allows any of the above policies to be combined flexibly.

In addition to the detailed pages on each policy, an introduction to the role of each policy in resilience engineering is also provided in the wiki.

Using Polly with HttpClient factory from ASPNET Core 2.1

For using Polly with HttpClient factory from ASPNET Core 2.1, see our detailed wiki page, then come back here or explore the wiki to learn more about the operation of each policy.

Release notes

- The change log describes changes by release.
- We tag Pull Requests and Issues with milestones which match to nuget package release numbers.
- Breaking changes are called out in the wiki (v7 ; v6) with simple notes on any necessary steps to upgrade.

Supported targets

For details of supported compilation targets by version, see the supported targets grid.

Role of the readme and the wiki

This ReadMe aims to give a quick overview of all Polly features - including enough to get you started with any policy. For deeper detail on any policy, and many other aspects of Polly, be sure also to check out the wiki documentation.

Usage – fault-handling, reactive policies

Fault-handling policies handle specific exceptions thrown by, or results returned by, the delegates you execute through the policy.

Step 1 : Specify the exceptions/faults you want the policy to handle

```
// Single exception type
Policy
  .Handle<HttpRequestException>()

// Single exception type with condition
Policy
  .Handle<SqlException>(ex => ex.Number == 1205)

// Multiple exception types
Policy
  .Handle<HttpRequestException>()
```

```
  .Or<OperationCanceledException>()

// Multiple exception types with condition
Policy
  .Handle<SqlException>(ex => ex.Number == 1205)
  .Or<ArgumentException>(ex => ex.ParamName == "example")

// Inner exceptions of ordinary exceptions or AggregateException, with or without co
// (HandleInner matches exceptions at both the top-level and inner exceptions)
Policy
  .HandleInner<HttpRequestException>()
  .OrInner<OperationCanceledException>(ex => ex.CancellationToken != myToken)
```

Step 1b: (optionally) Specify return results you want to handle

From Polly v4.3.0 onwards, policies wrapping calls returning a `TResult` can also handle `TResult` return values:

```
// Handle return value with condition
Policy
  .HandleResult<HttpResponseMessage>(r => r.StatusCode == HttpStatusCode.NotFound)

// Handle multiple return values
Policy
  .HandleResult<HttpResponseMessage>(r => r.StatusCode == HttpStatusCode.InternalSer
  .OrResult(r => r.StatusCode == HttpStatusCode.BadGateway)

// Handle primitive return values (implied use of .Equals())
Policy
  .HandleResult<HttpStatusCode>(HttpStatusCode.InternalServerError)
  .OrResult(HttpStatusCode.BadGateway)

// Handle both exceptions and return values in one policy
HttpStatusCode[] httpStatusCodesWorthRetrying = {
   HttpStatusCode.RequestTimeout, // 408
   HttpStatusCode.InternalServerError, // 500
   HttpStatusCode.BadGateway, // 502
   HttpStatusCode.ServiceUnavailable, // 503
   HttpStatusCode.GatewayTimeout // 504
};
HttpResponseMessage result = await Policy
  .Handle<HttpRequestException>()
  .OrResult<HttpResponseMessage>(r => httpStatusCodesWorthRetrying.Contains(r.Status
  .RetryAsync(...)
  .ExecuteAsync( /* some Func<Task<HttpResponseMessage>> */ )
```

For more information, see Handling Return Values at foot of this readme.

Step 2 : Specify how the policy should handle those faults

Retry

```
// Retry once
Policy
  .Handle<SomeExceptionType>()
  .Retry()

// Retry multiple times
Policy
  .Handle<SomeExceptionType>()
  .Retry(3)

// Retry multiple times, calling an action on each retry
// with the current exception and retry count
Policy
    .Handle<SomeExceptionType>()
    .Retry(3, onRetry: (exception, retryCount) =>
    {
        // Add logic to be executed before each retry, such as logging
    });

// Retry multiple times, calling an action on each retry
// with the current exception, retry count and context
// provided to Execute()
Policy
    .Handle<SomeExceptionType>()
    .Retry(3, onRetry: (exception, retryCount, context) =>
    {
        // Add logic to be executed before each retry, such as logging
    });
```

Retry forever (until succeeds)

```
// Retry forever
Policy
  .Handle<SomeExceptionType>()
  .RetryForever()

// Retry forever, calling an action on each retry with the
// current exception
Policy
```

```
  .Handle<SomeExceptionType>()
  .RetryForever(onRetry: exception =>
  {
        // Add logic to be executed before each retry, such as logging
  });

// Retry forever, calling an action on each retry with the
// current exception and context provided to Execute()
Policy
  .Handle<SomeExceptionType>()
  .RetryForever(onRetry: (exception, context) =>
  {
        // Add logic to be executed before each retry, such as logging
  });
```

Wait and retry

```
// Retry, waiting a specified duration between each retry.
// (The wait is imposed on catching the failure, before making the next try.)
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(new[]
  {
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3)
  });

// Retry, waiting a specified duration between each retry,
// calling an action on each retry with the current exception
// and duration
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(new[]
  {
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3)
  }, (exception, timeSpan) => {
    // Add logic to be executed before each retry, such as logging
  });

// Retry, waiting a specified duration between each retry,
// calling an action on each retry with the current exception,
// duration and context provided to Execute()
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(new[]
  {
```

```
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3)
  }, (exception, timeSpan, context) => {
    // Add logic to be executed before each retry, such as logging
  });

// Retry, waiting a specified duration between each retry,
// calling an action on each retry with the current exception,
// duration, retry count, and context provided to Execute()
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(new[]
  {
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3)
  }, (exception, timeSpan, retryCount, context) => {
    // Add logic to be executed before each retry, such as logging
  });

// Retry a specified number of times, using a function to
// calculate the duration to wait between retries based on
// the current retry attempt (allows for exponential backoff)
// In this case will wait for
//  2 ^ 1 = 2 seconds then
//  2 ^ 2 = 4 seconds then
//  2 ^ 3 = 8 seconds then
//  2 ^ 4 = 16 seconds then
//  2 ^ 5 = 32 seconds
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(5, retryAttempt =>
        TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
  );

// Retry a specified number of times, using a function to
// calculate the duration to wait between retries based on
// the current retry attempt, calling an action on each retry
// with the current exception, duration and context provided
// to Execute()
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(
    5,
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
    (exception, timeSpan, context) => {
      // Add logic to be executed before each retry, such as logging
    }
  );
```

```
// Retry a specified number of times, using a function to
// calculate the duration to wait between retries based on
// the current retry attempt, calling an action on each retry
// with the current exception, duration, retry count, and context
// provided to Execute()
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(
    5,
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
    (exception, timeSpan, retryCount, context) => {
      // Add logic to be executed before each retry, such as logging
    }
  );
```

The above code demonstrates how to build common wait-and-retry patterns from scratch, but our community also came up with an awesome contrib to wrap the common cases in helper methods: see Polly.Contrib.WaitAndRetry.

For `WaitAndRetry` policies handling Http Status Code 429 Retry-After, see wiki documentation.

Wait and retry forever (until succeeds)

```
// Wait and retry forever
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetryForever(retryAttempt =>
        TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
    );

// Wait and retry forever, calling an action on each retry with the
// current exception and the time to wait
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetryForever(
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
    (exception, timespan) =>
    {
        // Add logic to be executed before each retry, such as logging
    });

// Wait and retry forever, calling an action on each retry with the
// current exception, time to wait, and context provided to Execute()
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetryForever(
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
```

```
    (exception, timespan, context) =>
    {
        // Add logic to be executed before each retry, such as logging
    });
```

If all retries fail, a retry policy rethrows the final exception back to the calling code.

For more depth see also: Retry policy documentation on wiki.

Circuit Breaker

```
// Break the circuit after the specified number of consecutive exceptions
// and keep circuit broken for the specified duration.
Policy
    .Handle<SomeExceptionType>()
    .CircuitBreaker(2, TimeSpan.FromMinutes(1));

// Break the circuit after the specified number of consecutive exceptions
// and keep circuit broken for the specified duration,
// calling an action on change of circuit state.
Action<Exception, TimeSpan> onBreak = (exception, timespan) => { ... };
Action onReset = () => { ... };
CircuitBreakerPolicy breaker = Policy
    .Handle<SomeExceptionType>()
    .CircuitBreaker(2, TimeSpan.FromMinutes(1), onBreak, onReset);

// Break the circuit after the specified number of consecutive exceptions
// and keep circuit broken for the specified duration,
// calling an action on change of circuit state,
// passing a context provided to Execute().
Action<Exception, TimeSpan, Context> onBreak = (exception, timespan, context) => { .
Action<Context> onReset = context => { ... };
CircuitBreakerPolicy breaker = Policy
    .Handle<SomeExceptionType>()
    .CircuitBreaker(2, TimeSpan.FromMinutes(1), onBreak, onReset);

// Monitor the circuit state, for example for health reporting.
CircuitState state = breaker.CircuitState;

/*
CircuitState.Closed - Normal operation. Execution of actions allowed.
CircuitState.Open - The automated controller has opened the circuit. Execution of ac
CircuitState.HalfOpen - Recovering from open state, after the automated break durati
CircuitState.Isolated - Circuit held manually in an open state. Execution of actions
*/

// Manually open (and hold open) a circuit breaker - for example to manually isolate
breaker.Isolate();
```

```
// Reset the breaker to closed state, to start accepting actions again.
breaker.Reset();
```

Circuit-breaker policies block exceptions by throwing `BrokenCircuitException` when the circuit is broken. See: Circuit-Breaker documentation on wiki.

Note that circuit-breaker policies rethrow all exceptions, even handled ones. A circuit-breaker exists to measure faults and break the circuit when too many faults occur, but does not orchestrate retries. Combine a circuit-breaker with a retry policy as needed.

Advanced Circuit Breaker

```
// Break the circuit if, within any period of duration samplingDuration,
// the proportion of actions resulting in a handled exception exceeds failureThresho
// provided also that the number of actions through the circuit in the period
// is at least minimumThroughput.

Policy
    .Handle<SomeExceptionType>()
    .AdvancedCircuitBreaker(
        failureThreshold: 0.5, // Break on >=50% actions result in handled exception
        samplingDuration: TimeSpan.FromSeconds(10), // ... over any 10 second period
        minimumThroughput: 8, // ... provided at least 8 actions in the 10 second pe
        durationOfBreak: TimeSpan.FromSeconds(30) // Break for 30 seconds.
                );

// Configuration overloads taking state-change delegates are
// available as described for CircuitBreaker above.

// Circuit state monitoring and manual controls are
// available as described for CircuitBreaker above.
```

For more detail see: Advanced Circuit-Breaker documentation on wiki.

For more information on the Circuit Breaker pattern in general see:

- Making the Netflix API More Resilient
- Circuit Breaker (Martin Fowler)
- Circuit Breaker Pattern (Microsoft)
- Original Circuit Breaking Link

Fallback

```
// Provide a substitute value, if an execution faults.
Policy<UserAvatar>
   .Handle<FooException>()
   .OrResult(null)
   .Fallback<UserAvatar>(UserAvatar.Blank)

// Specify a func to provide a substitute value, if execution faults.
Policy<UserAvatar>
   .Handle<FooException>()
   .OrResult(null)
   .Fallback<UserAvatar>(() => UserAvatar.GetRandomAvatar()) // where: public UserAv

// Specify a substitute value or func, calling an action (eg for logging) if the fal
Policy<UserAvatar>
   .Handle<FooException>()
   .Fallback<UserAvatar>(UserAvatar.Blank, onFallback: (exception, context) =>
    {
        // Add extra logic to be called when the fallback is invoked, such as loggin
    });
```

For more detail see: Fallback policy documentation on wiki.

Step 3 : Execute code through the policy

Execute an `Action`, `Func`, or lambda delegate equivalent, through the policy. The policy governs execution of the code passed to the `.Execute()` (or similar) method.

> *Note: The code examples below show defining the policy and executing code through it in the same scope, for simplicity. See the notes after the code examples for other usage patterns.*

```
// Execute an action
var policy = Policy
             .Handle<SomeExceptionType>()
             .Retry();

policy.Execute(() => DoSomething());

// Execute an action passing arbitrary context data
var policy = Policy
    .Handle<SomeExceptionType>()
    .Retry(3, (exception, retryCount, context) =>
    {
        var methodThatRaisedException = context["methodName"];
                Log(exception, methodThatRaisedException);
```

```
    });

policy.Execute(
        () => DoSomething(),
        new Dictionary<string, object>() {{ "methodName", "some method" }}
);

// Execute a function returning a result
var policy = Policy
              .Handle<SomeExceptionType>()
              .Retry();

var result = policy.Execute(() => DoSomething());

// Execute a function returning a result passing arbitrary context data
var policy = Policy
    .Handle<SomeExceptionType>()
    .Retry(3, (exception, retryCount, context) =>
    {
        object methodThatRaisedException = context["methodName"];
        Log(exception, methodThatRaisedException)
    });

var result = policy.Execute(
    () => DoSomething(),
    new Dictionary<string, object>() {{ "methodName", "some method" }}
);

// You can of course chain it all together
Policy
  .Handle<SqlException>(ex => ex.Number == 1205)
  .Or<ArgumentException>(ex => ex.ParamName == "example")
  .Retry()
  .Execute(() => DoSomething());
```

Richer policy consumption patterns

Defining and consuming the policy in the same scope, as shown above, is the most immediate way to use Polly. Consider also:

- Separate policy definition from policy consumption, and inject policies into the code which will consume them. This enables many unit-testing scenarios.
- If your application uses Polly in a number of locations, define all policies at startup, and place them in a `PolicyRegistry`. This is a common pattern in .NET Core applications. For instance, you might define your own extension method on `IServiceCollection` to configure the policies you will consume elsewhere in the application. PolicyRegistry also combines well with DI to support unit-testing.

```
public static ConfigurePollyPolicies(this IServiceCollection services)
{
    PolicyRegistry registry = new PolicyRegistry()
    {
        { "RepositoryResilienceStrategy", /* define Policy or PolicyWrap strategy */
        { "CollaboratingMicroserviceResilienceStrategy", /* define Policy or PolicyW
        { "ThirdPartyApiResilienceStrategy", /* define Policy or PolicyWrap strategy
        /* etc */
    };

    services.AddSingleton<IReadOnlyPolicyRegistry<string>>(registry);
}
```

Usage – proactive policies

The proactive policies add resilience strategies that are not based on handling faults which the governed code may throw or return.

Step 1 : Configure

Timeout

Optimistic timeout

Optimistic timeout operates via CancellationToken and assumes delegates you execute support co-operative cancellation. You must use `Execute/Async(...)` overloads taking a `CancellationToken`, and the executed delegate must honor that `CancellationToken`.

```
// Timeout and return to the caller after 30 seconds, if the executed delegate has n
Policy
  .Timeout(30)

// Configure timeout as timespan.
Policy
  .Timeout(TimeSpan.FromMilliseconds(2500))

// Configure variable timeout via a func provider.
Policy
  .Timeout(() => myTimeoutProvider)) // Func<TimeSpan> myTimeoutProvider

// Timeout, calling an action if the action times out
Policy
  .Timeout(30, onTimeout: (context, timespan, task) =>
```

```
    {
        // Add extra logic to be invoked when a timeout occurs, such as logging
    });

// Eg timeout, logging that the execution timed out:
Policy
  .Timeout(30, onTimeout: (context, timespan, task) =>
    {
        logger.Warn($"{context.PolicyKey} at {context.OperationKey}: execution timed
    });

// Eg timeout, capturing any exception from the timed-out task when it completes:
Policy
  .Timeout(30, onTimeout: (context, timespan, task) =>
    {
        task.ContinueWith(t => {
            if (t.IsFaulted) logger.Error($"{context.PolicyKey} at {context.Operatio
        });
    });
```

Example execution:

```
Policy timeoutPolicy = Policy.TimeoutAsync(30);
HttpResponseMessage httpResponse = await timeoutPolicy
    .ExecuteAsync(
      async ct => await httpClient.GetAsync(endpoint, ct), // Execute a delegate whi
      CancellationToken.None // In this case, CancellationToken.None is passed into
      );
```

Pessimistic timeout

Pessimistic timeout allows calling code to 'walk away' from waiting for an executed delegate to complete, even if it does not support cancellation. In synchronous executions this is at the expense of an extra thread; see deep doco on wiki for more detail.

```
// Timeout after 30 seconds, if the executed delegate has not completed.  Enforces t
Policy
  .Timeout(30, TimeoutStrategy.Pessimistic)

// (All syntax variants outlined for optimistic timeout above also exist for pessimi
```

Example execution:

```
Policy timeoutPolicy = Policy.TimeoutAsync(30, TimeoutStrategy.Pessimistic);
var response = await timeoutPolicy
    .ExecuteAsync(
      async () => await FooNotHonoringCancellationAsync(), // Execute a delegate whi
      );
```

Timeout policies throw `TimeoutRejectedException` when timeout occurs.

For more detail see: Timeout policy documentation on wiki.

Bulkhead

```
// Restrict executions through the policy to a maximum of twelve concurrent actions.
Policy
  .Bulkhead(12)

// Restrict executions through the policy to a maximum of twelve concurrent actions,
// with up to two actions waiting for an execution slot in the bulkhead if all slots
Policy
  .Bulkhead(12, 2)

// Restrict concurrent executions, calling an action if an execution is rejected
Policy
  .Bulkhead(12, context =>
    {
        // Add callback logic for when the bulkhead rejects execution, such as loggi
    });

// Monitor the bulkhead available capacity, for example for health/load reporting.
var bulkhead = Policy.Bulkhead(12, 2);
// ...
int freeExecutionSlots = bulkhead.BulkheadAvailableCount;
int freeQueueSlots     = bulkhead.QueueAvailableCount;
```

Bulkhead policies throw `BulkheadRejectedException` if items are queued to the bulkhead when the bulkhead execution and queue are both full.

For more detail see: Bulkhead policy documentation on wiki.

Cache

```
var memoryCache = new MemoryCache(new MemoryCacheOptions());
var memoryCacheProvider = new MemoryCacheProvider(memoryCache);
var cachePolicy = Policy.Cache(memoryCacheProvider, TimeSpan.FromMinutes(5));
```

```
// For .NET Core DI examples see the CacheProviders linked to from https://github.co
// - https://github.com/App-vNext/Polly.Caching.MemoryCache
// - https://github.com/App-vNext/Polly.Caching.IDistributedCache

// Define a cache policy with absolute expiration at midnight tonight.
var cachePolicy = Policy.Cache(memoryCacheProvider, new AbsoluteTtl(DateTimeOffset.N

// Define a cache policy with sliding expiration: items remain valid for another 5 m
var cachePolicy = Policy.Cache(memoryCacheProvider, new SlidingTtl(TimeSpan.FromMinu

// Define a cache Policy, and catch any cache provider errors for logging.
var cachePolicy = Policy.Cache(myCacheProvider, TimeSpan.FromMinutes(5),
   (context, key, ex) => {
       logger.Error($"Cache provider, for key {key}, threw exception: {ex}."); // (f
   }
);

// Execute through the cache as a read-through cache: check the cache first; if not
// The key to use for caching, for a particular execution, is specified by setting t
// Example: "FooKey" is the cache key that will be used in the below execution.
TResult result = cachePolicy.Execute(context => getFoo(), new Context("FooKey"));
```

For richer options and details of using further cache providers see: Cache policy documentation on wiki.

PolicyWrap

```
// Define a combined policy strategy, built of previously-defined policies.
var policyWrap = Policy
  .Wrap(fallback, cache, retry, breaker, timeout, bulkhead);
// (wraps the policies around any executed delegate: fallback outermost ... bulkhead
policyWrap.Execute(...)

// Define a standard resilience strategy ...
PolicyWrap commonResilience = Policy.Wrap(retry, breaker, timeout);

// ... then wrap in extra policies specific to a call site, at that call site:
Avatar avatar = Policy<UserAvatar>
   .Handle<FooException>()
   .Fallback<Avatar>(Avatar.Blank)
   .Wrap(commonResilience)
   .Execute(() => { /* get avatar */ });

// Share commonResilience, but wrap different policies in at another call site:
Reputation reps = Policy
   .Handle<FooException>()
   .Fallback<Reputation>(Reputation.NotAvailable)
```

```
    .Wrap(commonResilience)
    .Execute(() => { /* get reputation */ });
```

For more detail see: PolicyWrap documentation on wiki.

NoOp

```
// Define a policy which will simply cause delegates passed for execution to be exec
// This is useful for stubbing-out Polly in unit tests,
// or in application situations where your code architecture might expect a policy
// but you simply want to pass the execution through without policy intervention.
NoOpPolicy noOp = Policy.NoOp();
```

For more detail see: NoOp documentation on wiki.

Step 2 : Execute the policy

As for fault-handling policies above.

Post-execution: capturing the result, or any final exception

Using the `ExecuteAndCapture(...)` methods you can capture the outcome of an execution: the methods return a `PolicyResult` instance which describes whether the outcome was a successful execution or a fault.

```
var policyResult = await Policy
              .Handle<HttpRequestException>()
              .RetryAsync()
              .ExecuteAndCaptureAsync(() => DoSomethingAsync());
/*
policyResult.Outcome - whether the call succeeded or failed
policyResult.FinalException - the final exception captured, will be null if the call
policyResult.ExceptionType - was the final exception an exception the policy was def
policyResult.Result - if executing a func, the result if the call succeeded or the t
*/
```

Handing return values, and Policy<TResult>

As described at step 1b, from Polly v4.3.0 onwards, policies can handle return values and exceptions in combination:

```
// Handle both exceptions and return values in one policy
HttpStatusCode[] httpStatusCodesWorthRetrying = {
   HttpStatusCode.RequestTimeout, // 408
   HttpStatusCode.InternalServerError, // 500
   HttpStatusCode.BadGateway, // 502
   HttpStatusCode.ServiceUnavailable, // 503
   HttpStatusCode.GatewayTimeout // 504
};
HttpResponseMessage result = await Policy
  .Handle<HttpRequestException>()
  .OrResult<HttpResponseMessage>(r => httpStatusCodesWorthRetrying.Contains(r.Status
  .RetryAsync(...)
  .ExecuteAsync( /* some Func<Task<HttpResponseMessage>> */ )
```

The exceptions and return results to handle can be expressed fluently in any order.

Strongly-typed Policy<TResult>

Configuring a policy with `.HandleResult<TResult>(...)` or `.OrResult<TResult>(...)` generates a strongly-typed `Policy<TResult>` of the specific policy type, eg `Retry<TResult>`, `AdvancedCircuitBreaker<TResult>`.

These policies must be used to execute delegates returning `TResult`, ie:

- `Execute(Func<TResult>)` (and related overloads)
- `ExecuteAsync(Func<CancellationToken, Task<TResult>>)` (and related overloads)

ExecuteAndCapture<TResult>()

`.ExecuteAndCapture(...)` on non-generic policies returns a `PolicyResult` with properties:

```
policyResult.Outcome - whether the call succeeded or failed
policyResult.FinalException - the final exception captured; will be null if the
call succeeded
policyResult.ExceptionType - was the final exception an exception the policy was
defined to handle (like HttpRequestException above) or an unhandled one (say
Exception)? Will be null if the call succeeded.
```

```
policyResult.Result - if executing a func, the result if the call succeeded;
otherwise, the type's default value
```

`.ExecuteAndCapture<TResult>(Func<TResult>)` on strongly-typed policies adds two properties:

```
policyResult.FaultType - was the final fault handled an exception or a result
handled by the policy? Will be null if the delegate execution succeeded.
policyResult.FinalHandledResult - the final fault result handled; will be null or
the type's default value, if the call succeeded
```

State-change delegates on Policy<TResult> policies

In non-generic policies handling only exceptions, state-change delegates such as `onRetry` and `onBreak` take an `Exception` parameter.

In generic-policies handling `TResult` return values, state-change delegates are identical except they take a `DelegateResult<TResult>` parameter in place of `Exception`. `DelegateResult<TResult>` has two properties:

- `Exception // The exception just thrown if policy is in process of handling an exception (otherwise null)`
- `Result // The TResult just raised, if policy is in process of handling a result (otherwise default(TResult))`

BrokenCircuitException<TResult>

Non-generic CircuitBreaker policies throw a `BrokenCircuitException` when the circuit is broken. This `BrokenCircuitException` contains the last exception (the one which caused the circuit to break) as the `InnerException`.

For `CircuitBreakerPolicy<TResult>` policies:

- A circuit broken due to an exception throws a `BrokenCircuitException` with `InnerException` set to the exception which triggered the break (as previously).
- A circuit broken due to handling a result throws a `BrokenCircuitException<TResult>` with the `Result` property set to the result which caused the circuit to break.

Policy Keys and Context data

```
// Identify policies with a PolicyKey, using the WithPolicyKey() extension method
// (for example, for correlation in logs or metrics)

var policy = Policy
    .Handle<DataAccessException>()
    .Retry(3, onRetry: (exception, retryCount, context) =>
       {
           logger.Error($"Retry {retryCount} of {context.PolicyKey} at {context.Oper
       })
    .WithPolicyKey("MyDataAccessPolicy");

// Identify call sites with an OperationKey, by passing in a Context
var customerDetails = policy.Execute(myDelegate, new Context("GetCustomerDetails"));

// "MyDataAccessPolicy" -> context.PolicyKey
// "GetCustomerDetails  -> context.OperationKey


// Pass additional custom information from call site into execution context
var policy = Policy
    .Handle<DataAccessException>()
    .Retry(3, onRetry: (exception, retryCount, context) =>
       {
           logger.Error($"Retry {retryCount} of {context.PolicyKey} at {context.Oper
       })
    .WithPolicyKey("MyDataAccessPolicy");

int id = ... // customer id from somewhere
var customerDetails = policy.Execute(context => GetCustomer(id),
    new Context("GetCustomerDetails", new Dictionary<string, object>() {{"Type","Cus
    ));
```

For more detail see: Keys and Context Data on wiki.

PolicyRegistry

```
// Create a policy registry (for example on application start-up)
PolicyRegistry registry = new PolicyRegistry();

// Populate the registry with policies
registry.Add("StandardHttpResilience", myStandardHttpResiliencePolicy);
// Or:
registry["StandardHttpResilience"] = myStandardHttpResiliencePolicy;

// Pass the registry instance to usage sites by DI, perhaps
public class MyServiceGateway
```

```
{
    public void MyServiceGateway(..., IReadOnlyPolicyRegistry<string> registry, ...)
    {
        ...
    }
}
// (Or if you prefer ambient-context pattern, use a thread-safe singleton)

// Use a policy from the registry
registry.Get<IAsyncPolicy<HttpResponseMessage>>("StandardHttpResilience")
    .ExecuteAsync<HttpResponseMessage>(...)
```

`PolicyRegistry` has a range of further dictionary-like semantics such as `.ContainsKey(...)`, `.TryGet<TPolicy>(...)`, `.Count`, `.Clear()`, and `Remove(...)`.

Available from v5.2.0. For more detail see: PolicyRegistry on wiki.

Asynchronous Support

Polly fully supports asynchronous executions, using the asynchronous methods:

- `RetryAsync`
- `WaitAndRetryAsync`
- `CircuitBreakerAsync`
- (etc)
- `ExecuteAsync`
- `ExecuteAndCaptureAsync`

In place of their synchronous counterparts:

- `Retry`
- `WaitAndRetry`
- `CircuitBreaker`
- (etc)
- `Execute`
- `ExecuteAndCapture`

Async overloads exist for all policy types and for all `Execute()` and `ExecuteAndCapture()` overloads.

Usage example:

```
await Policy
  .Handle<SqlException>(ex => ex.Number == 1205)
  .Or<ArgumentException>(ex => ex.ParamName == "example")
  .RetryAsync()
  .ExecuteAsync(() => DoSomethingAsync());
```

SynchronizationContext

Async continuations and retries by default do not run on a captured synchronization context. To change this, use `.ExecuteAsync(...)` overloads taking a boolean `continueOnCapturedContext` parameter.

Cancellation support

Async policy execution supports cancellation via `.ExecuteAsync(...)` overloads taking a `CancellationToken`.

The token you pass as the `cancellationToken` parameter to the `ExecuteAsync(...)` call serves three purposes:

- It cancels Policy actions such as further retries, waits between retries or waits for a bulkhead execution slot.
- It is passed by the policy as the `CancellationToken` input parameter to any delegate executed through the policy, to support cancellation during delegate execution.
- In common with the Base Class Library implementation in `Task.Run(…)` and elsewhere, if the cancellation token is cancelled before execution begins, the user delegate is not executed at all.

```
// Try several times to retrieve from a uri, but support cancellation at any time.
CancellationToken cancellationToken = // ...
var policy = Policy
    .Handle<HttpRequestException>()
    .WaitAndRetryAsync(new[] {
        TimeSpan.FromSeconds(1),
        TimeSpan.FromSeconds(2),
        TimeSpan.FromSeconds(4)
    });
var response = await policy.ExecuteAsync(ct => httpClient.GetAsync(uri, ct), cancell
```

From Polly v5.0, synchronous executions also support cancellation via `CancellationToken`.

Thread safety

All Polly policies are fully thread-safe. You can safely re-use policies at multiple call sites, and execute through policies concurrently on different threads.

While the internal operation of the policy is thread-safe, this does not magically make delegates you execute through the policy thread-safe: if delegates you execute through the policy are not thread-safe, they remain not thread-safe.

Interfaces

Polly v5.2.0 adds interfaces intended to support `PolicyRegistry` and to group Policy functionality by the interface segregation principle. Polly's interfaces are not intended for coding your own policy implementations against.

Execution interfaces: `ISyncPolicy` etc

Execution interfaces `ISyncPolicy`, `IAsyncPolicy`, `ISyncPolicy<TResult>` and `IAsyncPolicy<TResult>` define the execution overloads available to policies targeting sync/async, and non-generic / generic calls respectively.

Policy-kind interfaces: `ICircuitBreakerPolicy` etc

Orthogonal to the execution interfaces, interfaces specific to the kind of Policy define properties and methods common to that type of policy.

For example, `ICircuitBreakerPolicy` defines

- `CircuitState CircuitState`
- `Exception LastException`
- `void Isolate()`
- `void Reset()`

with `ICircuitBreakerPolicy<TResult> : ICircuitBreakerPolicy` adding:

- `TResult LastHandledResult`.

This allows collections of similar kinds of policy to be treated as one - for example, for monitoring all your circuit-breakers as described here.

For more detail see: Polly and interfaces on wiki.

Simmy

Simmy is a major new companion project adding a chaos-engineering and fault-injection dimension to Polly, through the provision of policies to selectively inject faults or latency.

Head over to the Simmy repo to find out more.

Custom policies

From Polly v7.0 it is possible to create your own custom policies outside Polly. These custom policies can integrate in to all the existing goodness from Polly: the `Policy.Handle<>()` syntax; PolicyWrap; all the execution-dispatch overloads.

For more info see our blog series:

- Part I: Introducing custom Polly policies and the Polly.Contrib
- Part II: Authoring a non-reactive custom policy (a policy which acts on all executions)
- Part III: Authoring a reactive custom policy (a policy which react to faults).
- Part IV: Custom policies for all execution types: sync and async, generic and non-generic.

We provide a starter template for a custom policy for developing your own custom policy.

Polly-Contrib

Polly now has a Polly-Contrib to allow the community to contribute policies or other enhancements around Polly with a low burden of ceremony.

Have a contrib you'd like to publish under Polly-Contrib? Contact us with an issue here or on Polly slack, and we can set up a CI-ready Polly.Contrib repo to which you have full rights, to help you manage and deliver your awesomeness to the community!

We also provide:

- a blank starter template for a custom policy (see above for more on custom policies)
- a template repo for any other contrib

Both templates contain a full project structure referencing Polly, Polly's default build targets, and a build to build and test your contrib and make a nuget package.

Available via Polly-Contrib

- Polly.Contrib.WaitAndRetry: a collection of concise helper methods for common wait-and-retry strategies; and a new jitter formula combining exponential backoff with a very even distribution of randomly-jittered retry intervals.
- Polly.Contrib.AzureFunctions.CircuitBreaker: a distributed circuit-breaker implemented in Azure Functions; consumable in Azure Functions, or from anywhere over http.
- Simmy: our chaos engineering project.
- Polly.Contrib.TimingPolicy: a starter policy to publish execution timings of any call executed through Policy.
- Polly.Contrib.LoggingPolicy: a policy simply to log handled exceptions/faults, and rethrow or bubble the fault outwards.

3rd Party Libraries and Contributions

- Fluent Assertions - A set of .NET extension methods that allow you to more naturally specify the expected outcome of a TDD or BDD-style test | Apache License 2.0 (Apache)
- xUnit.net - Free, open source, community-focused unit testing tool for the .NET Framework | Apache License 2.0 (Apache)
- Ian Griffith's TimedLock
- Steven van Deursen's ReadOnlyDictionary (until Polly v5.0.6)
- Stephen Cleary's AsyncEx library for AsyncSemaphore (supports BulkheadAsync policy for .NET4.0 only) (until Polly v5.9.0) | MIT license
- @theraot's ExceptionDispatchInfo implementation for .NET4.0 (supports Timeout policy for .NET4.0 only) (until Polly v5.9.0) including also a contribution by @migueldeicaza | Licensed under and distributed under Creative Commons Attribution Share Alike license per StackExchange Terms of Service
- Build powered by Cake and GitVersionTask. Developers powered by Resharper, with thanks to JetBrains for OSS licensing.

Acknowledgements

- lokad-shared-libraries - Helper assemblies originally for .NET 3.5 and Silverlight 2.0 which were developed as part of the Open Source effort by Lokad.com (discontinued) | New BSD License
- @michael-wolfenden - The creator and mastermind of Polly!
- @ghuntley - Portable Class Library implementation.
- @mauricedb - Initial async implementation.
- @robgibbens - Added existing async files to PCL project
- Hacko - Added extra `NotOnCapturedContext` call to prevent potential deadlocks when blocking on asynchronous calls
- @ThomasMentzel - Added ability to capture the results of executing a policy via `ExecuteAndCapture`
- @yevhen - Added full control of whether to continue on captured synchronization context or not
- @reisenberger - Added full async cancellation support
- @reisenberger - Added async support for ContextualPolicy
- @reisenberger - Added ContextualPolicy support for circuit-breaker
- @reisenberger - Extended circuit-breaker for public monitoring and control
- @reisenberger - Added ExecuteAndCapture support with arbitrary context data
- @kristianhald and @reisenberger - Added AdvancedCircuitBreaker
- @reisenberger - Allowed async onRetry delegates to async retry policies
- @Lumirris - Add new Polly.Net40Async project/package supporting async for .NET40 via Microsoft.Bcl.Async
- @SteveCote - Added overloads to WaitAndRetry and WaitAndRetryAsync methods that accept an onRetry delegate which includes the attempt count.
- @reisenberger - Allowed policies to handle returned results; added strongly-typed policies Policy<TResult>;.
- @christopherbahr - Added optimisation for circuit-breaker hot path.
- @Finity - Fixed circuit-breaker threshold bug.
- @reisenberger - Add some missing ExecuteAndCapture/Async overloads.
- @brunolauze - Add CancellationToken support to synchronous executions (to support TimeoutPolicy).
- @reisenberger - Add PolicyWrap.
- @reisenberger - Add Fallback policy.
- @reisenberger - Add PolicyKeys and context to all policy executions, as bedrock for policy events and metrics tracking executions.
- @reisenberger, and contributions from @brunolauze - Add Bulkhead Isolation policy.

- @reisenberger - Add Timeout policy.
- @reisenberger - Fix .NETStandard 1.0 targeting. Remove PCL259 target. PCL259 support is provided via .NETStandard1.0 target, going forward.
- @reisenberger - Fix CircuitBreaker HalfOpen state and cases when breakDuration is shorter than typical call timeout. Thanks to @vgouw and @kharos for the reports and insightful thinking.
- @lakario - Tidy CircuitBreaker LastException property.
- @lakario - Add NoOpPolicy.
- @Julien-Mialon - Fixes, support and examples for .NETStandard compatibility with Xamarin PCL projects
- @reisenberger - Add mutable Context and extra overloads taking Context. Allows different parts of a policy execution to exchange data via the mutable Context travelling with each execution.
- @ankitbko - Add PolicyRegistry for storing and retrieving policies.
- @reisenberger - Add interfaces by policy type and execution type.
- @seanfarrow - Add IReadOnlyPolicyRegistry interface.
- @kesmy - Migrate solution to msbuild15, banish project.json and packages.config
- @hambudi - Ensure sync TimeoutPolicy with TimeoutStrategy.Pessimistic rethrows delegate exceptions without additional AggregateException.
- @jiimaho and @Extremo75 - Provide public factory methods for PolicyResult, to support testing.
- @Extremo75 - Allow fallback delegates to take handled fault as input parameter.
- @reisenberger and @seanfarrow - Add CachePolicy, with interfaces for pluggable cache providers and serializers.
- Thanks to the awesome devs at @tretton37 who delivered the following as part of a one-day in-company hackathon led by @reisenberger, sponsored by @tretton37 and convened by @thecodejunkie
  - @matst80 - Allow WaitAndRetry to take handled fault as an input to the sleepDurationProvider, allowing WaitAndRetry to take account of systems which specify a duration to wait as part of a fault response; eg Azure CosmosDB may specify this in `x-ms-retry-after-ms` headers or in a property to an exception thrown by the Azure CosmosDB SDK.
  - @MartinSStewart - Add GetPolicies() extension methods to IPolicyWrap.
  - @jbergens37 - Parallelize test running where possible, to improve overall build speed.
- @reisenberger - Add new .HandleInner(...) syntax for handling inner exceptions natively.

- @rjongeneelen and @reisenberger - Allow PolicyWrap configuration to configure policies via interfaces.
- @reisenberger - Performance improvements.
- @awarrenlove - Add ability to calculate cache Ttl based on item to cache.
- @erickhouse - Add a new onBreak overload that provides the prior state on a transition to an open state.
- @benagain - Bug fix: RelativeTtl in CachePolicy now always returns a ttl relative to time item is cached.
- @urig - Allow TimeoutPolicy to be configured with Timeout.InfiniteTimeSpan.
- @reisenberger - Integration with IHttpClientFactory for ASPNET Core 2.1.
- @freakazoid182 - WaitAnd/RetryForever overloads where onRetry takes the retry number as a parameter.
- @dustyhoppe - Overloads where onTimeout takes thrown exception as a parameter.
- @flin-zap - Catch missing async continuation control.
- @reisenberger - Clarify separation of sync and async policies.
- @reisenberger - Enable extensibility by custom policies hosted external to Polly.
- @seanfarrow - Enable collection initialization syntax for PolicyRegistry.
- @moerwald - Code clean-ups, usage of more concise C# members.
- @cmeeren - Enable cache policies to cache values of default(TResult).
- @aprooks - Build script tweaks for Mac and mono.
- @kesmy - Add Soucelink support, clean up cake build.
- @simluk - Fix continueOnCaptureContext not being honored in async retry implementation (bug in v7.1.0 only).
- @jnyrup - Upgrade tests to Fluent Assertions v5.9.0
- @SimonCropp - Add netcoreapp3.0 target; code clean-ups.
- @aerotog and @reisenberger - IConcurrentPolicyRegistry methods on PolicyRegistry

Sample Projects

- Polly-Samples contains practical examples for using various implementations of Polly. Please feel free to contribute to the Polly-Samples repository in order to assist others who are either learning Polly for the first time, or are seeking advanced examples and novel approaches provided by our generous community.

- Microsoft's eShopOnContainers project is a sample project demonstrating a .NET Microservices architecture and using Polly for resilience

Instructions for Contributing

Please be sure to branch from the head of the latest vX.Y.Z dev branch (rather than master) when developing contributions.

For github workflow, check out our Wiki. We are following the excellent GitHub Flow process, and would like to make sure you have all of the information needed to be a world-class contributor!

Since Polly is part of the .NET Foundation, we ask our contributors to abide by their Code of Conduct. To contribute (beyond trivial typo corrections), review and sign the .Net Foundation Contributor License Agreement. This ensures the community is free to use your contributions. The registration process can be completed entirely online.

Also, we've stood up a Slack channel for easier real-time discussion of ideas and the general direction of Polly as a whole. Be sure to join the conversation today!

License

Blogs, podcasts, courses, ebooks, architecture samples and videos around Polly

When we discover an interesting write-up on Polly, we'll add it to this list. If you have a blog post you'd like to share, please submit a PR!

Blog posts

- Try .NET Samples of Polly, the .NET Resilience Framework - by Bryan Hogan
- Create exceptional interactive documentation with Try .NET - The Polly NuGet library did! - by Scott Hanselman (writing about the work of Bryan Hogan)
- Adding resilience and Transient Fault handling to your .NET Core HttpClient with Polly - by Scott Hanselman
- Reliable Event Processing in Azure Functions - by Jeff Hollan

- Optimally configuring ASPNET Core HttpClientFactory including with Polly policies - by Muhammad Rehan Saeed
- Integrating HttpClientFactory with Polly for transient fault handling - by Steve Gordon
- Resilient network connectivity in Xamarin Forms - by Adam Pedley
- Policy recommendations for Azure Cognitive Services - by Joel Hulen
- Using Polly with F# async workflows - by Mark Seemann
- Building resilient applications with Polly (with focus on Azure SQL transient errors) - by Geovanny Alzate Sandoval
- Azure SQL transient errors - by Mattias Karlsson
- Polly series on No Dogma blog - by Bryan Hogan
- Polly 5.0 - a wider resilience framework! - by Dylan Reisenberger
- Implementing the retry pattern in c sharp using Polly - by Alastair Crabtree
- NuGet Package of the Week: Polly wanna fluently express transient exception handling policies in .NET? - by Scott Hanselman
- Exception handling policies with Polly - by Mark Timmings
- When you use the Polly circuit-breaker, make sure you share your Policy instances! - by Andrew Lock
- Polly is Repetitive, and I love it! - by Joel Hulen
- Using the Context to Obtain the Retry Count for Diagnostics - by Steve Gordon
- Passing an ILogger to Polly Policies - by Steve Gordon
- Using Polly and Flurl to improve your website - by Jeremy Lindsay.
- Exploring the Polly.Contrib.WaitAndRetry helpers - by Ben Hyrman, who also wrote most of the Polly.Contrib.WaitAndRetry documentation.

Podcasts

- June 2018: DotNetRocks features Polly as Carl Franklin and Richard Campbell chat with Dylan Reisenberger about policy patterns, and the new ASP NET Core 2.1 integration with IHttpClientFactory.

- April 2017: Dylan Reisenberger sits down virtually with Bryan Hogan of NoDogmaBlog for an Introduction to Polly podcast. Why do I need Polly? History of the Polly project. What do we mean by resilience and transient faults? How retry and circuit-breaker help. Exponential backoff. Stability patterns. Bulkhead isolation. Future directions (as at April 2017).

PluralSight course

- Bryan Hogan of the NoDogmaBlog has authored a PluralSight course on Polly. The course takes you through all the major features of Polly, with an additional module added in the fall of 2018 on Http Client Factory. The course examples are based around using Polly for fault tolerance when calling remote web services, but the principles and techniques are applicable to any context in which Polly may be used.

Sample microservices architecture and ebook

Sample microservices architecture

- Cesar de la Torre produced the Microsoft eShopOnContainers project, a sample project demonstrating a .NET Microservices architecture. The project uses Polly retry and circuit-breaker policies for resilience in calls to microservices, and in establishing connections to transports such as RabbitMQ.

ebook

- Accompanying the project is a .Net Microservices Architecture ebook with an extensive section (section 8) on using Polly for resilience, to which Dylan Reisenberger contributed. The ebook and code is now (June 2018) updated for using the latest ASP NET Core 2.1 features, Polly with IHttpClientFactory.

Twitter

- Follow Dylan Reisenberger on twitter for notification of new Polly releases, advance notice of new proposals, tweets of interesting resilience articles (no junk).

Videos

- Robust Applications with Polly, the .NET Resilience Framework, Bryan Hogan introduces Polly and explains how to use it to build a fault tolerant application.
- From MVP Houssem Dellai, a youtube video on How to use Polly with Xamarin Apps, covering wait-and-retry and discussing circuit-breaker policy with a demonstration in Xamarin Forms. Here is the source code of the application demonstrated in the video. Draws on the `ResilientHttpClient` from Microsoft's eShopOnContainers project.
- In the video, .NET Rocks Live with Jon Skeet and Bill Wagner, Bill Wagner discusses Polly.
- Scott Allen discusses Polly during his Building for Resiliency and Scale in the Cloud presentation at NDC.

- ASP.NET Community Standup April 24, 2018: Damian Edwards, Jon Galloway and Scott Hanselman discuss Scott Hanselman's blog on Polly with IHttpClientFactory and the Polly team documentation on IHttpClientFactory. Interesting background discussion also on feature richness and the importance of good documentation.

Releases 26

7.2.2+9 Latest
on Apr 11

+ 25 releases

Packages

No packages published

Used by 10.9k

+ 10,916

Contributors 57

+ 46 contributors

Languages

C# 99.9% Other 0.1%