Retry pattern

06/23/2017 • 10 minutes to read • Contributors all

In this article

Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. This can improve the stability of the application.

Context and problem

An application that communicates with elements running in the cloud has to be sensitive to the transient faults that can occur in this environment. Faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that occur when a service is busy.

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay it's likely to be successful. For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay it might succeed.

Solution

In the cloud, transient faults aren't uncommon and an application should be designed to handle them elegantly and transparently. This minimizes the effects faults can have on the business tasks the application is performing.

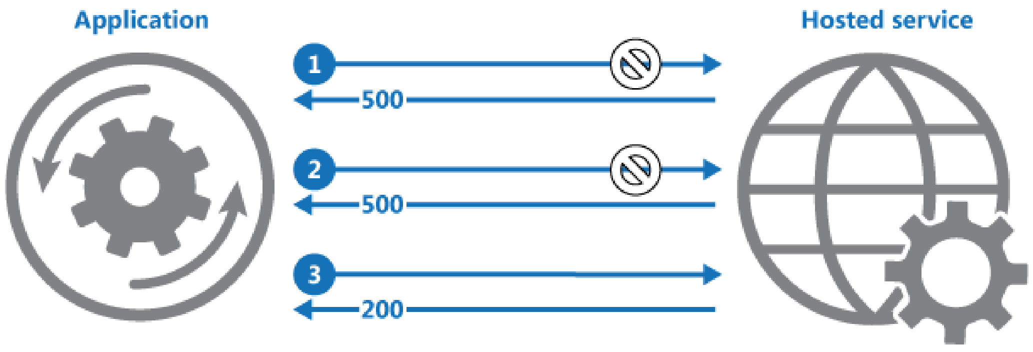
If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

- **Cancel**. If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
- **Retry**. If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated and the request will probably be successful.
- **Retry after delay**. If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time before retrying the request.

For the more common transient failures, the period between retries should be chosen to spread requests from multiple instances of the application as evenly as possible. This reduces the chance of a busy service continuing to be overloaded. If many instances of an application are continually overwhelming a service with retry requests, it'll take the service longer to recover.

If the request still fails, the application can wait and make another attempt. If necessary, this process can be repeated with increasing delays between retry attempts, until some maximum number of requests have been attempted. The delay can be increased incrementally or exponentially, depending on the type of failure and the probability that it'll be corrected during this time.

The following diagram illustrates invoking an operation in a hosted service using this pattern. If the request is unsuccessful after a predefined number of attempts, the application should treat the fault as an exception and handle it accordingly.



1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

The application should wrap all attempts to access a remote service in code that implements a retry policy matching one of the strategies listed above. Requests sent to different services can be subject to different policies. Some vendors provide libraries that implement retry policies, where the application can specify the maximum number of retries, the time between retry attempts, and other parameters.

An application should log the details of faults and failing operations. This information is useful to operators. If a service is frequently unavailable or busy, it's often because the service has exhausted its resources. You can reduce the frequency of these faults by scaling out the service. For example, if a database service is continually overloaded, it might be beneficial to partition the database and spread the load across multiple servers.

> Microsoft Entity Framework provides facilities for retrying database operations. Also, most Azure services and client SDKs include a retry mechanism. For more information, see Retry guidance for specific services.

Issues and considerations

You should consider the following points when deciding how to implement this pattern.

The retry policy should be tuned to match the business requirements of the application and the nature of the failure. For some noncritical operations, it's better to fail fast rather than retry several times and impact the throughput of the application. For example, in an interactive web application accessing a remote service, it's better to fail after a smaller number of retries with only a short delay between retry attempts, and display a suitable message to the user (for example, “please try again later”). For a batch application, it might be more appropriate to increase the number of retry attempts with an exponentially increasing delay between attempts.

An aggressive retry policy with minimal delay between attempts, and a large number of retries, could further degrade a busy service that's running close to or at capacity. This retry policy could also affect the responsiveness of the application if it's continually trying to perform a failing operation.

If a request still fails after a significant number of retries, it's better for the application to prevent further requests going to the same resource and simply report a failure immediately. When the period expires, the application can tentatively allow one or more requests through to see whether they're successful. For more details of this strategy, see the Circuit Breaker pattern.

Consider whether the operation is idempotent. If so, it's inherently safe to retry. Otherwise, retries could cause the operation to be executed more than once, with unintended side effects. For example, a service might receive the request, process the request successfully, but fail to send a response. At that point, the retry logic might re-send the request, assuming that the first request wasn't received.

A request to a service can fail for a variety of reasons raising different exceptions depending on the nature of the failure. Some exceptions indicate a failure that can be resolved quickly, while others indicate that the failure is longer lasting. It's useful for the retry policy to adjust the time between retry attempts based on the type of the exception.

Consider how retrying an operation that's part of a transaction will affect the overall transaction consistency. Fine tune the retry policy for transactional operations to maximize the chance of success and reduce the need to undo all the transaction steps.

Ensure that all retry code is fully tested against a variety of failure conditions. Check that it doesn't severely impact the performance or reliability of the application, cause excessive load on services and resources, or generate race conditions or bottlenecks.

Implement retry logic only where the full context of a failing operation is understood. For example, if a task that contains a retry policy invokes another task that also contains a retry policy, this extra layer of retries can add long delays to the processing. It might be better to configure the lower-level task to fail fast and report the reason for the failure back to the task that invoked it. This higher-level task can then handle the failure based on its own policy.

It's important to log all connectivity failures that cause a retry so that underlying problems with the application, services, or resources can be identified.

Investigate the faults that are most likely to occur for a service or a resource to discover if they're likely to be long lasting or terminal. If they are, it's better to handle the fault as an exception. The application can report or log the exception, and then try to continue either by invoking an alternative service (if one is available), or by offering degraded functionality. For more information on how to detect and handle long-lasting faults, see the Circuit Breaker pattern.

When to use this pattern

Use this pattern when an application could experience transient faults as it interacts with a remote service or accesses a remote resource. These faults are expected to be short lived, and repeating a request that has previously failed could succeed on a subsequent attempt.

This pattern might not be useful:

- When a fault is likely to be long lasting, because this can affect the responsiveness of an application. The application might be wasting time and resources trying to repeat a request that's likely to fail.
- For handling failures that aren't due to transient faults, such as internal exceptions caused by errors in the business logic of an application.
- As an alternative to addressing scalability issues in a system. If an application experiences frequent busy faults, it's often a sign that the service or resource being accessed should be scaled up.

Example

This example in C# illustrates an implementation of the Retry pattern. The `OperationWithBasicRetryAsync` method, shown below, invokes an external service asynchronously through the `TransientOperationAsync` method. The details of the `TransientOperationAsync` method will be specific to the service and are omitted from the sample code.

C# Copy

```csharp
private int retryCount = 3;
private readonly TimeSpan delay = TimeSpan.FromSeconds(5);

public async Task OperationWithBasicRetryAsync()
{
  int currentRetry = 0;

  for (;;)
  {
    try
    {
      // Call external service.
      await TransientOperationAsync();

      // Return or break.
      break;
    }
    catch (Exception ex)
    {
      Trace.TraceError("Operation Exception");

      currentRetry++;

      // Check if the exception thrown was a transient exception
      // based on the logic in the error detection strategy.
      // Determine whether to retry the operation, as well as how
      // long to wait, based on the retry strategy.
      if (currentRetry > this.retryCount || !IsTransient(ex))
      {
        // If this isn't a transient error or we shouldn't retry,
        // rethrow the exception.
        throw;
      }
    }

    // Wait to retry the operation.
    // Consider calculating an exponential delay here and
    // using a strategy best suited for the operation and fault.
    await Task.Delay(delay);
  }
}

// Async method that wraps a call to a remote service (details not shown).
private async Task TransientOperationAsync()
{
  ...
}
```

The statement that invokes this method is contained in a try/catch block wrapped in a for loop. The for loop exits if the call to the `TransientOperationAsync` method succeeds without throwing an exception. If the `TransientOperationAsync` method fails, the catch block examines the reason for the failure. If it's believed to be a transient error the code waits for a short delay before retrying the operation.

The for loop also tracks the number of times that the operation has been attempted, and if the code fails three times the exception is assumed to be more long lasting. If the exception isn't transient or it's long lasting, the catch handler throws an exception. This exception exits the for loop and should be caught by the code that invokes the `OperationWithBasicRetryAsync` method.

The `IsTransient` method, shown below, checks for a specific set of exceptions that are relevant to the environment the code is run in. The definition of a transient exception will vary according to the resources being accessed and the environment the operation is being performed in.

C# Copy

```
private bool IsTransient(Exception ex)
{
  // Determine if the exception is transient.
  // In some cases this is as simple as checking the exception type, in other
  // cases it might be necessary to inspect other properties of the exception.
  if (ex is OperationTransientException)
    return true;

  var webException = ex as WebException;
  if (webException != null)
  {
    // If the web exception contains one of the following status values
    // it might be transient.
    return new[] {WebExceptionStatus.ConnectionClosed,
                  WebExceptionStatus.Timeout,
                  WebExceptionStatus.RequestCanceled }.
            Contains(webException.Status);
  }

  // Additional exception checking logic goes here.
  return false;
}
```

Related patterns and guidance

- Circuit Breaker pattern. The Retry pattern is useful for handling transient faults. If a failure is expected to be more long lasting, it might be more appropriate to implement the Circuit Breaker pattern. The Retry pattern can also be used in conjunction with a circuit breaker to provide a comprehensive approach to handling faults.
- Retry guidance for specific services
- Connection Resiliency