Retry guidance for specific services

08/13/2018 • 40 minutes to read • Contributors all

In this article

Azure Active Directory
Cosmos DB
Event Hubs
IoT Hub
Azure Redis Cache
Azure Search
Service Bus
Service Fabric
SQL Database using ADO.NET
SQL Database using Entity Framework 6
SQL Database using Entity Framework Core
Azure Storage
General REST and retry guidelines

Most Azure services and client SDKs include a retry mechanism. However, these differ because each service has different characteristics and requirements, and so each retry mechanism is tuned to a specific service. This guide summarizes the retry mechanism features for the majority of Azure services, and includes information to help you use, adapt, or extend the retry mechanism for that service.

For general guidance on handling transient faults, and retrying connections and operations against services and resources, see Retry guidance.

The following table summarizes the retry features for the Azure services described in this guidance.

Service	Retry capabilities	Policy configuration	Scope	Telemetry features
Azure Active Directory	Native in ADAL library	Embedded into ADAL library	Internal	None
Cosmos DB	Native in service	Non-configurable	Global	TraceSource
Data Lake Store	Native in client	Non-configurable	Individual operations	None
Event Hubs	Native in client	Programmatic	Client	None
IoT Hub	Native in client SDK	Programmatic	Client	None
Redis Cache	Native in client	Programmatic	Client	TextWriter
Search	Native in client	Programmatic	Client	ETW or Custom

Service	Retry capabilities	Policy configuration	Scope	Telemetry features
Service Bus	Native in client	Programmatic	Namespace Manager, Messaging Factory, and Client	ETW
Service Fabric	Native in client	Programmatic	Client	None
SQL Database with ADO.NET	Polly	Declarative and programmatic	Single statements or blocks of code	Custom
SQL Database with Entity Framework	Native in client	Programmatic	Global per AppDomain	None
SQL Database with Entity Framework Core	Native in client	Programmatic	Global per AppDomain	None
Storage	Native in client	Programmatic	Client and individual operations	TraceSource

> **Note**
>
> For most of the Azure built-in retry mechanisms, there is currently no way apply a different retry policy for different types of error or exception. You should configure a policy that provides the optimum average performance and availability. One way to fine-tune the policy is to analyze log files to determine the type of transient faults that are occurring.

Azure Active Directory

Azure Active Directory (Azure AD) is a comprehensive identity and access management cloud solution that combines core directory services, advanced identity governance, security, and application access management. Azure AD also offers developers an identity management platform to deliver access control to their applications, based on centralized policy and rules.

> **Note**
>
> For retry guidance on Managed Service Identity endpoints, see How to use an Azure VM Managed Service Identity (MSI) for token acquisition.

Retry mechanism

There is a built-in retry mechanism for Azure Active Directory in the Active Directory Authentication Library (ADAL). To avoid unexpected lockouts, we recommend that third-party libraries and application code do **not** retry failed connections, but allow ADAL to handle retries.

Retry usage guidance

Consider the following guidelines when using Azure Active Directory:

- When possible, use the ADAL library and the built-in support for retries.
- If you are using the REST API for Azure Active Directory, retry the operation if the result code is 429 (Too Many Requests) or an error in the 5xx range. Do not retry for any other errors.

- An exponential back-off policy is recommended for use in batch scenarios with Azure Active Directory.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Sample target E2E max latency	Retry strategy	Settings	Values	How it works
Interactive, UI, or foreground	2 sec	FixedInterval	Retry count Retry interval First fast retry	3 500 ms true	Attempt 1 - delay 0 sec Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	60 sec	ExponentialBackoff	Retry count Min back-off Max back-off Delta back-off First fast retry	5 0 sec 60 sec 2 sec false	Attempt 1 - delay 0 sec Attempt 2 - delay ~2 sec Attempt 3 - delay ~6 sec Attempt 4 - delay ~14 sec Attempt 5 - delay ~30 sec

More information

- Azure Active Directory Authentication Libraries

Cosmos DB

Cosmos DB is a fully managed multi-model database that supports schemaless JSON data. It offers configurable and reliable performance, native JavaScript transactional processing, and is built for the cloud with elastic scale.

Retry mechanism

The `DocumentClient` class automatically retries failed attempts. To set the number of retries and the maximum wait time, configure ConnectionPolicy.RetryOptions. Exceptions that the client raises are either beyond the retry policy or are not transient errors.

If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `DocumentClientException`.

Policy configuration

The following table shows the default settings for the `RetryOptions` class.

Setting	Default value	Description
MaxRetryAttemptsOnThrottledRequests	9	The maximum number of retries if the request fails because Cosmos DB applied rate limiting on the client.
MaxRetryWaitTimeInSeconds	30	The maximum retry time in seconds.

Example

C# Copy

```
DocumentClient client = new DocumentClient(new Uri(endpoint), authKey); ;
var options = client.ConnectionPolicy.RetryOptions;
options.MaxRetryAttemptsOnThrottledRequests = 5;
options.MaxRetryWaitTimeInSeconds = 15;
```

Telemetry

Retry attempts are logged as unstructured trace messages through a .NET **TraceSource**. You must configure a **TraceListener** to capture the events and write them to a suitable destination log.

For example, if you add the following to your App.config file, traces will be generated in a text file in the same location as the executable:

XML Copy

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="SourceSwitch" value="Verbose"/>
    </switches>
    <sources>
      <source name="DocDBTrace" switchName="SourceSwitch" switchType="System.Diagnostics.-
SourceSwitch" >
        <listeners>
          <add name="MyTextListener" type="System.Diagnostics.TextWriterTraceListener" trace-
OutputOptions="DateTime,ProcessId,ThreadId" initializeData="CosmosDBTrace.txt"></add>
        </listeners>
      </source>
    </sources>
  </system.diagnostics>
</configuration>
```

Event Hubs

Azure Event Hubs is a hyper-scale telemetry ingestion service that collects, transforms, and stores millions of events.

Retry mechanism

Retry behavior in the Azure Event Hubs Client Library is controlled by the `RetryPolicy` property on the `EventHubClient` class. The default policy retries with exponential backoff when Azure Event Hub returns a transient `EventHubsException` or an `OperationCanceledException`.

Example

C# Copy

```
EventHubClient client = EventHubClient.CreateFromConnectionString("[event_hub_connec-
tion_string]");
client.RetryPolicy = RetryPolicy.Default;
```

More information

.NET Standard client library for Azure Event Hubs

IoT Hub

Azure IoT Hub is a service for connecting, monitoring, and managing devices to develop Internet of Things (IoT) applications.

Retry mechanism

The Azure IoT device SDK can detect errors in the network, protocol, or application. Based on the error type, the SDK checks whether a retry needs to be performed. If the error is *recoverable*, the SDK begins to retry using the configured retry policy.

The default retry policy is *exponential back-off with random jitter*, but it can be configured.

Policy configuration

Policy configuration differs by language. For more details, see IoT Hub retry policy configuration.

More information

- IoT Hub retry policy
- Troubleshoot IoT Hub device disconnection

Azure Redis Cache

Azure Redis Cache is a fast data access and low latency cache service based on the popular open source Redis Cache. It is secure, managed by Microsoft, and is accessible from any application in Azure.

The guidance in this section is based on using the StackExchange.Redis client to access the cache. A list of other suitable clients can be found on the Redis website, and these may have different retry mechanisms.

Note that the StackExchange.Redis client uses multiplexing through a single connection. The recommended usage is to create an instance of the client at application startup and use this instance for all operations against the cache. For this reason, the connection to the cache is made only once, and so all of the guidance in this section is related to the retry policy for this initial connection—and not for each operation that accesses the cache.

Retry mechanism

The StackExchange.Redis client uses a connection manager class that is configured through a set of options, including:

- **ConnectRetry**. The number of times a failed connection to the cache will be retried.
- **ReconnectRetryPolicy**. The retry strategy to use.
- **ConnectTimeout**. The maximum waiting time in milliseconds.

Policy configuration

Retry policies are configured programmatically by setting the options for the client before connecting to the cache. This can be done by creating an instance of the **ConfigurationOptions** class, populating its properties, and passing it to the **Connect** method.

The built-in classes support linear (constant) delay and exponential backoff with randomized retry intervals. You can also create a custom retry policy by implementing the **IReconnectRetryPolicy** interface.

The following example configures a retry strategy using exponential backoff.

C# Copy

```
var deltaBackOffInMilliseconds = TimeSpan.FromSeconds(5).Milliseconds;
var maxDeltaBackOffInMilliseconds = TimeSpan.FromSeconds(20).Milliseconds;
var options = new ConfigurationOptions
{
    EndPoints = {"localhost"},
    ConnectRetry = 3,
    ReconnectRetryPolicy = new ExponentialRetry(deltaBackOffInMilliseconds, maxDeltaBackOffIn-
Milliseconds),
```

```
    ConnectTimeout = 2000
};
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

Alternatively, you can specify the options as a string, and pass this to the **Connect** method. The **ReconnectRetryPolicy** property cannot be set this way, only through code.

C# Copy

```
var options = "localhost,connectRetry=3,connectTimeout=2000";
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

You can also specify options directly when you connect to the cache.

C# Copy

```
var conn = ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,connectRetry=3");
```

For more information, see Stack Exchange Redis Configuration in the StackExchange.Redis documentation.

The following table shows the default settings for the built-in retry policy.

Context	Setting	Default value (v 1.2.2)	Meaning
ConfigurationOptions	ConnectRetry	3	The number of times to repeat connect attempts during the initial connection operation.
	ConnectTimeout	Maximum 5000 ms plus SyncTimeout	Timeout (ms) for connect operations. Not a delay between retry attempts.
	SyncTimeout	1000	Time (ms) to allow for synchronous operations.
	ReconnectRetryPolicy	LinearRetry 5000 ms	Retry every 5000 ms.

> **Note**
>
> For synchronous operations, `SyncTimeout` can add to the end-to-end latency, but setting the value too low can cause excessive timeouts. See How to troubleshoot Azure Redis Cache. In general, avoid using synchronous operations, and use asynchronous operations instead. For more information, see Pipelines and Multiplexers.

Retry usage guidance

Consider the following guidelines when using Azure Redis Cache:

- The StackExchange Redis client manages its own retries, but only when establishing a connection to the cache when the application first starts. You can configure the connection timeout, the number of retry attempts, and the time between retries to establish this connection, but the retry policy does not apply to operations against the cache.
- Instead of using a large number of retry attempts, consider falling back by accessing the original data source instead.

Telemetry

You can collect information about connections (but not other operations) using a **TextWriter**.

C# Copy

```
var writer = new StringWriter();
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

An example of the output this generates is shown below.

text Copy

```
localhost:6379,connectTimeout=2000,connectRetry=3
1 unique nodes specified
Requesting tie-break from localhost:6379 > __Booksleeve_TieBreak...
Allowing endpoints 00:00:02 to respond...
localhost:6379 faulted: SocketFailure on PING
localhost:6379 failed to nominate (Faulted)
> UnableToResolvePhysicalConnection on GET
No masters detected
localhost:6379: Standalone v2.0.0, master; keep-alive: 00:01:00; int: Connecting; sub: Connect-
ing; not in use: DidNotRespond
localhost:6379: int ops=0, qu=0, qs=0, qc=1, wr=0, sync=1, socks=2; sub ops=0, qu=0, qs=0,
qc=0, wr=0, socks=2
Circular op-count snapshot; int: 0 (0.00 ops/s; spans 10s); sub: 0 (0.00 ops/s; spans 10s)
Sync timeouts: 0; fire and forget: 0; last heartbeat: -1s ago
resetting failing connections to retry...
retrying; attempts left: 2...
...
```

Examples

The following code example configures a constant (linear) delay between retries when initializing the StackExchange.Redis client. This example shows how to set the configuration using a **ConfigurationOptions** instance.

C# Copy

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    var retryTimeInMilliseconds = TimeSpan.FromSeconds(4).Milliseconds; // de-
lay between retries

                    // Using object-based configuration.
                    var options = new ConfigurationOptions
                                  {
                                      EndPoints = { "localhost" },
                                      ConnectRetry = 3,
                                      ReconnectRetryPolicy = new LinearRetry(retryTimeIn-
Milliseconds)
                                  };
                    ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options,
writer);
```

```
                    // Store a reference to the multiplexer for use in the application.
                }
                catch
                {
                    Console.WriteLine(writer.ToString());
                    throw;
                }
            }
        }
    }
}
```

The next example sets the configuration by specifying the options as a string. The connection timeout is the maximum period of time to wait for a connection to the cache, not the delay between retry attempts. Note that the **ReconnectRetryPolicy** property can only be set by code.

C# Copy

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    // Using string-based configuration.
                    var options = "localhost,connectRetry=3,connectTimeout=2000";
                    ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options,
writer);

                    // Store a reference to the multiplexer for use in the application.
                }
                catch
                {
                    Console.WriteLine(writer.ToString());
                    throw;
                }
            }
        }
    }
}
```

For more examples, see Configuration on the project website.

More information

- Redis website

Azure Search

Azure Search can be used to add powerful and sophisticated search capabilities to a website or application, quickly and easily tune search results, and construct rich and fine-tuned ranking models.

Retry mechanism

Retry behavior in the Azure Search SDK is controlled by the `SetRetryPolicy` method on the SearchServiceClient and SearchIndexClient classes. The default policy retries with exponential backoff when Azure Search returns a 5xx or 408 (Request Timeout) response.

Telemetry

Trace with ETW or by registering a custom trace provider. For more information, see the AutoRest documentation.

Service Bus

Service Bus is a cloud messaging platform that provides loosely coupled message exchange with improved scale and resiliency for components of an application, whether hosted in the cloud or on-premises.

Retry mechanism

Service Bus implements retries using implementations of the RetryPolicy base class. All of the Service Bus clients expose a **RetryPolicy** property that can be set to one of the implementations of the **RetryPolicy** base class. The built-in implementations are:

- The RetryExponential class. This exposes properties that control the back-off interval, the retry count, and the **TerminationTimeBuffer** property that is used to limit the total time for the operation to complete.
- The NoRetry class. This is used when retries at the Service Bus API level are not required, such as when retries are managed by another process as part of a batch or multistep operation.

Service Bus actions can return a range of exceptions, as listed in Service Bus messaging exceptions. The list provides information about which if these indicate that retrying the operation is appropriate. For example, a **ServerBusyException** indicates that the client should wait for a period of time, then retry the operation. The occurrence of a **ServerBusyException** also causes Service Bus to switch to a different mode, in which an extra 10-second delay is added to the computed retry delays. This mode is reset after a short period.

The exceptions returned from Service Bus expose the **IsTransient** property that indicates if the client should retry the operation. The built-in **RetryExponential** policy relies on the **IsTransient** property in the **MessagingException** class, which is the base class for all Service Bus exceptions. If you create custom implementations of the **RetryPolicy** base class you could use a combination of the exception type and the **IsTransient** property to provide more fine-grained control over retry actions. For example, you could detect a **QuotaExceededException** and take action to drain the queue before retrying sending a message to it.

Policy configuration

Retry policies are set programmatically, and can be set as a default policy for a **NamespaceManager** and for a **MessagingFactory**, or individually for each messaging client. To set the default retry policy for a messaging session you set the **RetryPolicy** of the **NamespaceManager**.

C# Copy

```
namespaceManager.Settings.RetryPolicy = new RetryExponential(minBackoff:
TimeSpan.FromSeconds(0.1),
                                                              maxBackoff: TimeSpan.FromSec-
onds(30),
                                                              maxRetryCount: 3);
```

To set the default retry policy for all clients created from a messaging factory, you set the **RetryPolicy** of the **MessagingFactory**.

C# Copy

```
messagingFactory.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                                    maxBackoff: TimeSpan.FromSeconds(30),
                                                    maxRetryCount: 3);
```

To set the retry policy for a messaging client, or to override its default policy, you set its **RetryPolicy** property using an instance of the required policy class:

C# Copy

```
client.RetryPolicy = new RetryExponential(minBackoff: TimeSpan.FromSeconds(0.1),
                                          maxBackoff: TimeSpan.FromSeconds(30),
                                          maxRetryCount: 3);
```

The retry policy cannot be set at the individual operation level. It applies to all operations for the messaging client. The following table shows the default settings for the built-in retry policy.

Setting	Default value	Meaning
Policy	Exponential	Exponential back-off.
MinimalBackoff	0	Minimum back-off interval. This is added to the retry interval computed from deltaBackoff.
MaximumBackoff	30 seconds	Maximum back-off interval. MaximumBackoff is used if the computed retry interval is greater than MaxBackoff.
DeltaBackoff	3 seconds	Back-off interval between retries. Multiples of this timespan will be used for subsequent retry attempts.
TimeBuffer	5 seconds	The termination time buffer associated with the retry. Retry attempts will be abandoned if the remaining time is less than TimeBuffer.
MaxRetryCount	10	The maximum number of retries.
ServerBusyBaseSleepTime	10 seconds	If the last exception encountered was **ServerBusyException**, this value will be added to the computed retry interval. This value cannot be changed.

Retry usage guidance

Consider the following guidelines when using Service Bus:

- When using the built-in **RetryExponential** implementation, do not implement a fallback operation as the policy reacts to Server Busy exceptions and automatically switches to an appropriate retry mode.
- Service Bus supports a feature called Paired Namespaces that implements automatic failover to a backup queue in a separate namespace if the queue in the primary namespace fails. Messages from the secondary queue can be sent back to the primary queue when it recovers. This feature helps to address transient failures. For more information, see Asynchronous Messaging Patterns and High Availability.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Example maximum latency	Retry policy	Settings	How it works

Context	Example maximum latency	Retry policy	Settings	How it works
Interactive, UI, or foreground	2 seconds*	Exponential	MinimumBackoff = 0 MaximumBackoff = 30 sec. DeltaBackoff = 300 msec. TimeBuffer = 300 msec. MaxRetryCount = 2	Attempt 1: Delay 0 sec. Attempt 2: Delay ~300 msec. Attempt 3: Delay ~900 msec.
Background or batch	30 seconds	Exponential	MinimumBackoff = 1 MaximumBackoff = 30 sec. DeltaBackoff = 1.75 sec. TimeBuffer = 5 sec. MaxRetryCount = 3	Attempt 1: Delay ~1 sec. Attempt 2: Delay ~3 sec. Attempt 3: Delay ~6 msec. Attempt 4: Delay ~13 msec.

* Not including additional delay that is added if a Server Busy response is received.

Telemetry

Service Bus logs retries as ETW events using an **EventSource**. You must attach an **EventListener** to the event source to capture the events and view them in Performance Viewer, or write them to a suitable destination log. The retry events are of the following form:

text Copy

```
Microsoft-ServiceBus-Client/RetryPolicyIteration
ThreadID="14,500"
FormattedMessage="[TrackingId:] RetryExponential: Operation Get:https://retry-tests.service-
bus.windows.net/TestQueue/?api-version=2014-05 at iteration 0 is retrying after
00:00:00.1000000 sleep because of Microsoft.ServiceBus.Messaging.MessagingCommunicationExcep-
tion: The remote name could not be resolved: 'retry-
tests.servicebus.windows.net'.TrackingId:6a26f99c-dc6d-422e-8565-f89fdd0d4fe3,
TimeStamp:9/5/2014 10:00:13 PM."
trackingId=""
policyType="RetryExponential"
operation="Get:https://retry-tests.servicebus.windows.net/TestQueue/?api-version=2014-05"
iteration="0"
iterationSleep="00:00:00.1000000"
lastExceptionType="Microsoft.ServiceBus.Messaging.MessagingCommunicationException"
exceptionMessage="The remote name could not be resolved: 'retry-
tests.servicebus.windows.net'.TrackingId:6a26f99c-dc6d-422e-8565-
f89fdd0d4fe3,TimeStamp:9/5/2014 10:00:13 PM"
```

Examples

The following code example shows how to set the retry policy for:

- A namespace manager. The policy applies to all operations on that manager, and cannot be overridden for individual operations.
- A messaging factory. The policy applies to all clients created from that factory, and cannot be overridden when creating individual clients.
- An individual messaging client. After a client has been created, you can set the retry policy for that client. The policy applies to all operations on that client.

C# Copy

```
using System;
using System.Threading.Tasks;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;

namespace RetryCodeSamples
{
    class ServiceBusCodeSamples
    {
        private const string connectionString =
            @"Endpoint=sb://[my-namespace].servicebus.windows.net/;
                SharedAccessKeyName=RootManageSharedAccessKey;
                SharedAccessKey=C99.........Mk=";

        public async static Task Samples()
        {
            const string QueueName = "TestQueue";

            ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;

            var namespaceManager = NamespaceManager.CreateFromConnectionString(connection-
String);

            // The namespace manager will have a default exponential policy with 10 retry at-
tempts
            // and a 3 second delay delta.
            // Retry delays will be approximately 0 sec, 3 sec, 9 sec, 25 sec and the fixed 30
sec,
            // with an extra 10 sec added when receiving a ServiceBusyException.

            {
                // Set different values for the retry policy, used for all operations on the
namespace manager.
                namespaceManager.Settings.RetryPolicy =
                    new RetryExponential(
                        minBackoff: TimeSpan.FromSeconds(0),
                        maxBackoff: TimeSpan.FromSeconds(30),
                        maxRetryCount: 3);

                // Policies cannot be specified on a per-operation basis.
                if (!await namespaceManager.QueueExistsAsync(QueueName))
                {
                    await namespaceManager.CreateQueueAsync(QueueName);
                }
            }

            var messagingFactory = MessagingFactory.Create(
                namespaceManager.Address, namespaceManager.Settings.TokenProvider);
            // The messaging factory will have a default exponential policy with 10 retry at-
tempts
            // and a 3 second delay delta.
            // Retry delays will be approximately 0 sec, 3 sec, 9 sec, 25 sec and the fixed 30
sec,
            // with an extra 10 sec added when receiving a ServiceBusyException.

            {
                // Set different values for the retry policy, used for clients created from it.
                messagingFactory.RetryPolicy =
                    new RetryExponential(
                        minBackoff: TimeSpan.FromSeconds(1),
                        maxBackoff: TimeSpan.FromSeconds(30),
                        maxRetryCount: 3);


                // Policies cannot be specified on a per-operation basis.
                var session = await messagingFactory.AcceptMessageSessionAsync();
            }
```

```
            {
                var client = messagingFactory.CreateQueueClient(QueueName);
                // The client inherits the policy from the factory that created it.


                // Set different values for the retry policy on the client.
                client.RetryPolicy =
                    new RetryExponential(
                        minBackoff: TimeSpan.FromSeconds(0.1),
                        maxBackoff: TimeSpan.FromSeconds(30),
                        maxRetryCount: 3);

                // Policies cannot be specified on a per-operation basis.
                var session = await client.AcceptMessageSessionAsync();
            }
        }
    }
}
```

More information

- Asynchronous messaging patterns and high availability

Service Fabric

Distributing reliable services in a Service Fabric cluster guards against most of the potential transient faults discussed in this article. Some transient faults are still possible, however. For example, the naming service might be in the middle of a routing change when it gets a request, causing it to throw an exception. If the same request comes 100 milliseconds later, it will probably succeed.

Internally, Service Fabric manages this kind of transient fault. You can configure some settings by using the `OperationRetrySettings` class while setting up your services. The following code shows an example. In most cases, this should not be necessary, and the default settings will be fine.

C# Copy

```
FabricTransportRemotingSettings transportSettings = new FabricTransportRemotingSettings
{
    OperationTimeout = TimeSpan.FromSeconds(30)
};

var retrySettings = new OperationRetrySettings(TimeSpan.FromSeconds(15),
TimeSpan.FromSeconds(1), 5);

var clientFactory = new FabricTransportServiceRemotingClientFactory(transportSettings);

var serviceProxyFactory = new ServiceProxyFactory((c) => clientFactory, retrySettings);

var client = serviceProxyFactory.CreateServiceProxy<ISomeService>(
    new Uri("fabric:/SomeApp/SomeStatefulReliableService"),
    new ServicePartitionKey(0));
```

More information

- Remote exception handling

SQL Database using ADO.NET

SQL Database is a hosted SQL database available in a range of sizes and as both a standard (shared) and premium (non-shared) service.

Retry mechanism

SQL Database has no built-in support for retries when accessed using ADO.NET. However, the return codes from requests can be used to determine why a request failed. For more information about SQL Database throttling, see Azure SQL Database resource limits. For a list of relevant error codes, see SQL error codes for SQL Database client applications.

You can use the Polly library to implement retries for SQL Database. See Transient fault handling with Polly.

Retry usage guidance

Consider the following guidelines when accessing SQL Database using ADO.NET:

- Choose the appropriate service option (shared or premium). A shared instance may suffer longer than usual connection delays and throttling due to the usage by other tenants of the shared server. If more predictable performance and reliable low latency operations are required, consider choosing the premium option.
- Ensure that you perform retries at the appropriate level or scope to avoid non-idempotent operations causing inconsistency in the data. Ideally, all operations should be idempotent so that they can be repeated without causing inconsistency. Where this is not the case, the retry should be performed at a level or scope that allows all related changes to be undone if one operation fails; for example, from within a transactional scope. For more information, see Cloud Service Fundamentals Data Access Layer – Transient Fault Handling.
- A fixed interval strategy is not recommended for use with Azure SQL Database except for interactive scenarios where there are only a few retries at very short intervals. Instead, consider using an exponential back-off strategy for the majority of scenarios.
- Choose a suitable value for the connection and command timeouts when defining connections. Too short a timeout may result in premature failures of connections when the database is busy. Too long a timeout may prevent the retry logic working correctly by waiting too long before detecting a failed connection. The value of the timeout is a component of the end-to-end latency; it is effectively added to the retry delay specified in the retry policy for every retry attempt.
- Close the connection after a certain number of retries, even when using an exponential back off retry logic, and retry the operation on a new connection. Retrying the same operation multiple times on the same connection can be a factor that contributes to connection problems. For an example of this technique, see Cloud Service Fundamentals Data Access Layer – Transient Fault Handling.
- When connection pooling is in use (the default) there is a chance that the same connection will be chosen from the pool, even after closing and reopening a connection. If this is the case, a technique to resolve it is to call the **ClearPool** method of the **SqlConnection** class to mark the connection as not reusable. However, you should do this only after several connection attempts have failed, and only when encountering the specific class of transient failures such as SQL timeouts (error code -2) related to faulty connections.
- If the data access code uses transactions initiated as **TransactionScope** instances, the retry logic should reopen the connection and initiate a new transaction scope. For this reason, the retryable code block should encompass the entire scope of the transaction.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Sample target E2E max latency	Retry strategy	Settings	Values	How it works
Interactive, UI, or foreground	2 sec	FixedInterval	Retry count Retry interval First fast retry	3 500 ms true	Attempt 1 - delay 0 sec Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms

Context	Sample target E2E max latency	Retry strategy	Settings	Values	How it works
Background or batch	30 sec	ExponentialBackoff	Retry count Min back-off Max back-off Delta back-off First fast retry	5 0 sec 60 sec 2 sec false	Attempt 1 - delay 0 sec Attempt 2 - delay ~2 sec Attempt 3 - delay ~6 sec Attempt 4 - delay ~14 sec Attempt 5 - delay ~30 sec

> **Note**
>
> The end-to-end latency targets assume the default timeout for connections to the service. If you specify longer connection timeouts, the end-to-end latency will be extended by this additional time for every retry attempt.

Examples

This section shows how you can use Polly to access Azure SQL Database using a set of retry policies configured in the `Policy` class.

The following code shows an extension method on the `SqlCommand` class that calls `ExecuteAsync` with exponential backoff.

C# Copy

```
public async static Task<SqlDataReader> ExecuteReaderWithRetryAsync(this SqlCommand command)
{
    GuardConnectionIsNotNull(command);

    var policy = Policy.Handle<Exception>().WaitAndRetryAsync(
        retryCount: 3, // Retry 3 times
        sleepDurationProvider: attempt => TimeSpan.FromMilliseconds(200 * Math.Pow(2, attempt -
1)), // Exponential backoff based on an initial 200 ms delay.
        onRetry: (exception, attempt) =>
        {
            // Capture some information for logging/telemetry.
            logger.LogWarn($"ExecuteReaderWithRetryAsync: Retry {attempt} due to
{exception}.");
        });

    // Retry the following call according to the policy.
    await policy.ExecuteAsync<SqlDataReader>(async token =>
    {
        // This code is executed within the Policy

        if (conn.State != System.Data.ConnectionState.Open) await conn.OpenAsync(token);
        return await command.ExecuteReaderAsync(System.Data.CommandBehavior.Default, token);

    }, cancellationToken);
}
```

This asynchronous extension method can be used as follows.

C# Copy

```
var sqlCommand = sqlConnection.CreateCommand();
sqlCommand.CommandText = "[some query]";

using (var reader = await sqlCommand.ExecuteReaderWithRetryAsync())
{
    // Do something with the values
}
```

More information

- Cloud Service Fundamentals Data Access Layer – Transient Fault Handling

For general guidance on getting the most from SQL Database, see Azure SQL Database performance and elasticity guide.

SQL Database using Entity Framework 6

SQL Database is a hosted SQL database available in a range of sizes and as both a standard (shared) and premium (non-shared) service. Entity Framework is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.

Retry mechanism

Retry support is provided when accessing SQL Database using Entity Framework 6.0 and higher through a mechanism called Connection resiliency / retry logic. The main features of the retry mechanism are:

- The primary abstraction is the **IDbExecutionStrategy** interface. This interface:
  - Defines synchronous and asynchronous **Execute*** methods.
  - Defines classes that can be used directly or can be configured on a database context as a default strategy, mapped to provider name, or mapped to a provider name and server name. When configured on a context, retries occur at the level of individual database operations, of which there might be several for a given context operation.
  - Defines when to retry a failed connection, and how.
- It includes several built-in implementations of the **IDbExecutionStrategy** interface:
  - Default - no retrying.
  - Default for SQL Database (automatic) - no retrying, but inspects exceptions and wraps them with suggestion to use the SQL Database strategy.
  - Default for SQL Database - exponential (inherited from base class) plus SQL Database detection logic.
- It implements an exponential back-off strategy that includes randomization.
- The built-in retry classes are stateful and are not thread-safe. However, they can be reused after the current operation is completed.
- If the specified retry count is exceeded, the results are wrapped in a new exception. It does not bubble up the current exception.

Policy configuration

Retry support is provided when accessing SQL Database using Entity Framework 6.0 and higher. Retry policies are configured programmatically. The configuration cannot be changed on a per-operation basis.

When configuring a strategy on the context as the default, you specify a function that creates a new strategy on demand. The following code shows how you can create a retry configuration class that extends the **DbConfiguration** base class.

C# Copy

```
public class BloggingContextConfiguration : DbConfiguration
{
  public BlogConfiguration()
  {
    // Set up the execution strategy for SQL Database (exponential) with 5 retries and 4 sec
delay
```

```
    this.SetExecutionStrategy(
        "System.Data.SqlClient", () => new SqlAzureExecutionStrategy(5,
TimeSpan.FromSeconds(4)));
  }
}
```

You can then specify this as the default retry strategy for all operations using the **SetConfiguration** method of the **DbConfiguration** instance when the application starts. By default, EF will automatically discover and use the configuration class.

C# Copy

```
DbConfiguration.SetConfiguration(new BloggingContextConfiguration());
```

You can specify the retry configuration class for a context by annotating the context class with a **DbConfigurationType** attribute. However, if you have only one configuration class, EF will use it without the need to annotate the context.

C# Copy

```
[DbConfigurationType(typeof(BloggingContextConfiguration))]
public class BloggingContext : DbContext
```

If you need to use different retry strategies for specific operations, or disable retries for specific operations, you can create a configuration class that allows you to suspend or swap strategies by setting a flag in the **CallContext**. The configuration class can use this flag to switch strategies, or disable the strategy you provide and use a default strategy. For more information, see Suspend Execution Strategy (EF6 onwards).

Another technique for using specific retry strategies for individual operations is to create an instance of the required strategy class and supply the desired settings through parameters. You then invoke its **ExecuteAsync** method.

C# Copy

```
var executionStrategy = new SqlAzureExecutionStrategy(5, TimeSpan.FromSeconds(4));
var blogs = await executionStrategy.ExecuteAsync(
    async () =>
    {
        using (var db = new BloggingContext("Blogs"))
        {
            // Acquire some values asynchronously and return them
        }
    },
    new CancellationToken()
);
```

The simplest way to use a **DbConfiguration** class is to locate it in the same assembly as the **DbContext** class. However, this is not appropriate when the same context is required in different scenarios, such as different interactive and background retry strategies. If the different contexts execute in separate AppDomains, you can use the built-in support for specifying configuration classes in the configuration file or set it explicitly using code. If the different contexts must execute in the same AppDomain, a custom solution will be required.

For more information, see Code-Based Configuration (EF6 onwards).

The following table shows the default settings for the built-in retry policy when using EF6.

Setting	Default value	Meaning
Policy	Exponential	Exponential back-off.

Setting	Default value	Meaning
MaxRetryCount	5	The maximum number of retries.
MaxDelay	30 seconds	The maximum delay between retries. This value does not affect how the series of delays are computed. It only defines an upper bound.
DefaultCoefficient	1 second	The coefficient for the exponential back-off computation. This value cannot be changed.
DefaultRandomFactor	1.1	The multiplier used to add a random delay for each entry. This value cannot be changed.
DefaultExponentialBase	2	The multiplier used to calculate the next delay. This value cannot be changed.

Retry usage guidance

Consider the following guidelines when accessing SQL Database using EF6:

- Choose the appropriate service option (shared or premium). A shared instance may suffer longer than usual connection delays and throttling due to the usage by other tenants of the shared server. If predictable performance and reliable low latency operations are required, consider choosing the premium option.
- A fixed interval strategy is not recommended for use with Azure SQL Database. Instead, use an exponential back-off strategy because the service may be overloaded, and longer delays allow more time for it to recover.
- Choose a suitable value for the connection and command timeouts when defining connections. Base the timeout on both your business logic design and through testing. You may need to modify this value over time as the volumes of data or the business processes change. Too short a timeout may result in premature failures of connections when the database is busy. Too long a timeout may prevent the retry logic working correctly by waiting too long before detecting a failed connection. The value of the timeout is a component of the end-to-end latency, although you cannot easily determine how many commands will execute when saving the context. You can change the default timeout by setting the **CommandTimeout** property of the **DbContext** instance.
- Entity Framework supports retry configurations defined in configuration files. However, for maximum flexibility on Azure you should consider creating the configuration programmatically within the application. The specific parameters for the retry policies, such as the number of retries and the retry intervals, can be stored in the service configuration file and used at runtime to create the appropriate policies. This allows the settings to be changed without requiring the application to be restarted.

Consider starting with the following settings for retrying operations. You cannot specify the delay between retry attempts (it is fixed and generated as an exponential sequence). You can specify only the maximum values, as shown here; unless you create a custom retry strategy. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Sample target E2E max latency	Retry policy	Settings	Values	How it works
Interactive, UI, or foreground	2 seconds	Exponential	MaxRetryCount MaxDelay	3 750 ms	Attempt 1 - delay 0 sec Attempt 2 - delay 750 ms Attempt 3 – delay 750 ms

Context	Sample target E2E max latency	Retry policy	Settings	Values	How it works
Background or batch	30 seconds	Exponential	MaxRetryCount MaxDelay	5 12 seconds	Attempt 1 - delay 0 sec Attempt 2 - delay ~1 sec Attempt 3 - delay ~3 sec Attempt 4 - delay ~7 sec Attempt 5 - delay 12 sec

> **Note**
>
> The end-to-end latency targets assume the default timeout for connections to the service. If you specify longer connection timeouts, the end-to-end latency will be extended by this additional time for every retry attempt.

Examples

The following code example defines a simple data access solution that uses Entity Framework. It sets a specific retry strategy by defining an instance of a class named **BlogConfiguration** that extends **DbConfiguration**.

C# Copy

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.SqlServer;
using System.Threading.Tasks;

namespace RetryCodeSamples
{
    public class BlogConfiguration : DbConfiguration
    {
        public BlogConfiguration()
        {
            // Set up the execution strategy for SQL Database (exponential) with 5 retries and
12 sec delay.
            // These values could be loaded from configuration rather than being hard-coded.
            this.SetExecutionStrategy(
                    "System.Data.SqlClient", () => new SqlAzureExecutionStrategy(5, TimeS-
pan.FromSeconds(12)));
        }
    }

    // Specify the configuration type if more than one has been defined.
    // [DbConfigurationType(typeof(BlogConfiguration))]
    public class BloggingContext : DbContext
    {
        // Definition of content goes here.
    }

    class EF6CodeSamples
    {
        public async static Task Samples()
        {
            // Execution strategy configured by DbConfiguration subclass, discovered automati-
cally or
            // or explicitly indicated through configuration or with an attribute. Default is
no retries.
            using (var db = new BloggingContext("Blogs"))
            {
                // Add, edit, delete blog items here, then:
                await db.SaveChangesAsync();
            }
        }
```

```
    }
}
```

More examples of using the Entity Framework retry mechanism can be found in Connection Resiliency / Retry Logic.

More information

- Azure SQL Database performance and elasticity guide

SQL Database using Entity Framework Core

Entity Framework Core is an object-relational mapper that enables .NET Core developers to work with data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write. This version of Entity Framework was written from the ground up, and doesn't automatically inherit all the features from EF6.x.

Retry mechanism

Retry support is provided when accessing SQL Database using Entity Framework Core through a mechanism called connection resiliency. Connection resiliency was introduced in EF Core 1.1.0.

The primary abstraction is the `IExecutionStrategy` interface. The execution strategy for SQL Server, including SQL Azure, is aware of the exception types that can be retried and has sensible defaults for maximum retries, delay between retries, and so on.

Examples

The following code enables automatic retries when configuring the DbContext object, which represents a session with the database.

C# Copy

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=EFMiscellanous.ConnectionResiliency;Trust-
ed_Connection=True;",
            options => options.EnableRetryOnFailure());
}
```

The following code shows how to execute a transaction with automatic retries, by using an execution strategy. The transaction is defined in a delegate. If a transient failure occurs, the execution strategy will invoke the delegate again.

C# Copy

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var transaction = db.Database.BeginTransaction())
        {
            db.Blogs.Add(new Blog { Url = "https://blogs.msdn.com/dotnet" });
            db.SaveChanges();

            db.Blogs.Add(new Blog { Url = "https://blogs.msdn.com/visualstudio" });
            db.SaveChanges();
```

```
            transaction.Commit();
        }
    });
}
```

Azure Storage

Azure Storage services include table and blob storage, files, and storage queues.

Retry mechanism

Retries occur at the individual REST operation level and are an integral part of the client API implementation. The client storage SDK uses classes that implement the IExtendedRetryPolicy Interface.

There are different implementations of the interface. Storage clients can choose from policies designed for accessing tables, blobs, and queues. Each implementation uses a different retry strategy that essentially defines the retry interval and other details.

The built-in classes provide support for linear (constant delay) and exponential with randomization retry intervals. There is also a no retry policy for use when another process is handling retries at a higher level. However, you can implement your own retry classes if you have specific requirements not provided by the built-in classes.

Alternate retries switch between primary and secondary storage service location if you are using read access geo-redundant storage (RA-GRS) and the result of the request is a retryable error. See Azure Storage Redundancy Options for more information.

Policy configuration

Retry policies are configured programmatically. A typical procedure is to create and populate a **TableRequestOptions**, **BlobRequestOptions**, **FileRequestOptions**, or **QueueRequestOptions** instance.

C# Copy

```
TableRequestOptions interactiveRequestOption = new TableRequestOptions()
{
  RetryPolicy = new LinearRetry(TimeSpan.FromMilliseconds(500), 3),
  // For Read-access geo-redundant storage, use PrimaryThenSecondary.
  // Otherwise set this to PrimaryOnly.
  LocationMode = LocationMode.PrimaryThenSecondary,
  // Maximum execution time based on the business use case.
  MaximumExecutionTime = TimeSpan.FromSeconds(2)
};
```

The request options instance can then be set on the client, and all operations with the client will use the specified request options.

C# Copy

```
client.DefaultRequestOptions = interactiveRequestOption;
var stats = await client.GetServiceStatsAsync();
```

You can override the client request options by passing a populated instance of the request options class as a parameter to operation methods.

C# Copy

```
var stats = await client.GetServiceStatsAsync(interactiveRequestOption, operationContext:
null);
```

You use an **OperationContext** instance to specify the code to execute when a retry occurs and when an operation has completed. This code can collect information about the operation for use in logs and telemetry.

C# Copy

```
// Set up notifications for an operation
var context = new OperationContext();
context.ClientRequestID = "some request id";
context.Retrying += (sender, args) =>
{
    /* Collect retry information */
};
context.RequestCompleted += (sender, args) =>
{
    /* Collect operation completion information */
};
var stats = await client.GetServiceStatsAsync(null, context);
```

In addition to indicating whether a failure is suitable for retry, the extended retry policies return a **RetryContext** object that indicates the number of retries, the results of the last request, whether the next retry will happen in the primary or secondary location (see table below for details). The properties of the **RetryContext** object can be used to decide if and when to attempt a retry. For more information, see IExtendedRetryPolicy.Evaluate Method.

The following tables show the default settings for the built-in retry policies.

Request options:

Setting	Default value	Meaning
MaximumExecutionTime	None	Maximum execution time for the request, including all potential retry attempts. If it is not specified, then the amount of time that a request is permitted to take is unlimited. In other words, the request might hang.
ServerTimeout	None	Server timeout interval for the request (value is rounded to seconds). If not specified, it will use the default value for all requests to the server. Usually, the best option is to omit this setting so that the server default is used.
LocationMode	None	If the storage account is created with the Read access geo-redundant storage (RA-GRS) replication option, you can use the location mode to indicate which location should receive the request. For example, if **PrimaryThenSecondary** is specified, requests are always sent to the primary location first. If a request fails, it is sent to the secondary location.
RetryPolicy	ExponentialPolicy	See below for details of each option.

Exponential policy:

Setting	Default value	Meaning
maxAttempt	3	Number of retry attempts.
deltaBackoff	4 seconds	Back-off interval between retries. Multiples of this timespan, including a random element, will be used for subsequent retry attempts.

Setting	Default value	Meaning
MinBackoff	3 seconds	Added to all retry intervals computed from deltaBackoff. This value cannot be changed.
MaxBackoff	120 seconds	MaxBackoff is used if the computed retry interval is greater than MaxBackoff. This value cannot be changed.

Linear policy:

Setting	Default value	Meaning
maxAttempt	3	Number of retry attempts.
deltaBackoff	30 seconds	Back-off interval between retries.

Retry usage guidance

Consider the following guidelines when accessing Azure storage services using the storage client API:

- Use the built-in retry policies from the Microsoft.Azure.Storage.RetryPolicies namespace where they are appropriate for your requirements. In most cases, these policies will be sufficient.
- Use the **ExponentialRetry** policy in batch operations, background tasks, or non-interactive scenarios. In these scenarios, you can typically allow more time for the service to recover—with a consequently increased chance of the operation eventually succeeding.
- Consider specifying the **MaximumExecutionTime** property of the **RequestOptions** parameter to limit the total execution time, but take into account the type and size of the operation when choosing a timeout value.
- If you need to implement a custom retry, avoid creating wrappers around the storage client classes. Instead, use the capabilities to extend the existing policies through the **IExtendedRetryPolicy** interface.
- If you are using read access geo-redundant storage (RA-GRS) you can use the **LocationMode** to specify that retry attempts will access the secondary read-only copy of the store should the primary access fail. However, when using this option you must ensure that your application can work successfully with data that may be stale if the replication from the primary store has not yet completed.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

Context	Sample target E2E max latency	Retry policy	Settings	Values	How it works
Interactive, UI, or foreground	2 seconds	Linear	maxAttempt deltaBackoff	3 500 ms	Attempt 1 - delay 500 ms Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	30 seconds	Exponential	maxAttempt deltaBackoff	5 4 seconds	Attempt 1 - delay ~3 sec Attempt 2 - delay ~7 sec Attempt 3 - delay ~15 sec

Telemetry

Retry attempts are logged to a **TraceSource**. You must configure a **TraceListener** to capture the events and write them to a suitable destination log. You can use the **TextWriterTraceListener** or **XmlWriterTraceListener** to write the data to a

log file, the **EventLogTraceListener** to write to the Windows Event Log, or the **EventProviderTraceListener** to write trace data to the ETW subsystem. You can also configure autoflushing of the buffer, and the verbosity of events that will be logged (for example, Error, Warning, Informational, and Verbose). For more information, see Client-side Logging with the .NET Storage Client Library.

Operations can receive an **OperationContext** instance, which exposes a **Retrying** event that can be used to attach custom telemetry logic. For more information, see OperationContext.Retrying Event.

Examples

The following code example shows how to create two **TableRequestOptions** instances with different retry settings; one for interactive requests and one for background requests. The example then sets these two retry policies on the client so that they apply for all requests, and also sets the interactive strategy on a specific request so that it overrides the default settings applied to the client.

C# Copy

```
using System;
using System.Threading.Tasks;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.RetryPolicies;
using Microsoft.WindowsAzure.Storage.Table;

namespace RetryCodeSamples
{
    class AzureStorageCodeSamples
    {
        private const string connectionString = "UseDevelopmentStorage=true";

        public async static Task Samples()
        {
            var storageAccount = CloudStorageAccount.Parse(connectionString);

            TableRequestOptions interactiveRequestOption = new TableRequestOptions()
            {
                RetryPolicy = new LinearRetry(TimeSpan.FromMilliseconds(500), 3),
                // For Read-access geo-redundant storage, use PrimaryThenSecondary.
                // Otherwise set this to PrimaryOnly.
                LocationMode = LocationMode.PrimaryThenSecondary,
                // Maximum execution time based on the business use case.
                MaximumExecutionTime = TimeSpan.FromSeconds(2)
            };

            TableRequestOptions backgroundRequestOption = new TableRequestOptions()
            {
                // Client has a default exponential retry policy with 4 sec delay and 3 retry
attempts
                // Retry delays will be approximately 3 sec, 7 sec, and 15 sec
                MaximumExecutionTime = TimeSpan.FromSeconds(30),
                // PrimaryThenSecondary in case of Read-access geo-redundant storage, else set
this to PrimaryOnly
                LocationMode = LocationMode.PrimaryThenSecondary
            };

            var client = storageAccount.CreateCloudTableClient();
            // Client has a default exponential retry policy with 4 sec delay and 3 retry at-
tempts
            // Retry delays will be approximately 3 sec, 7 sec, and 15 sec
            // ServerTimeout and MaximumExecutionTime are not set

            {
                // Set properties for the client (used on all requests unless overridden)
                // Different exponential policy parameters for background scenarios
                client.DefaultRequestOptions = backgroundRequestOption;
                // Linear policy for interactive scenarios
```

```
                client.DefaultRequestOptions = interactiveRequestOption;
            }

            {
                // set properties for a specific request
                var stats = await client.GetServiceStatsAsync(interactiveRequestOption, opera-
tionContext: null);
            }

            {
                // Set up notifications for an operation
                var context = new OperationContext();
                context.ClientRequestID = "some request id";
                context.Retrying += (sender, args) =>
                {
                    /* Collect retry information */
                };
                context.RequestCompleted += (sender, args) =>
                {
                    /* Collect operation completion information */
                };
                var stats = await client.GetServiceStatsAsync(null, context);
            }
        }
    }
}
```

More information

- Azure Storage client Library retry policy recommendations
- Storage Client Library 2.0 – Implementing retry policies

General REST and retry guidelines

Consider the following when accessing Azure or third-party services:

- Use a systematic approach to managing retries, perhaps as reusable code, so that you can apply a consistent methodology across all clients and all solutions.
- Consider using a retry framework such as Polly to manage retries if the target service or client has no built-in retry mechanism. This will help you implement a consistent retry behavior, and it may provide a suitable default retry strategy for the target service. However, you may need to create custom retry code for services that have nonstandard behavior, that do not rely on exceptions to indicate transient failures, or if you want to use a **Retry-Response** reply to manage retry behavior.
- The transient detection logic will depend on the actual client API you use to invoke the REST calls. Some clients, such as the newer **HttpClient** class, will not throw exceptions for completed requests with a non-success HTTP status code.
- The HTTP status code returned from the service can help to indicate whether the failure is transient. You may need to examine the exceptions generated by a client or the retry framework to access the status code or to determine the equivalent exception type. The following HTTP codes typically indicate that a retry is appropriate:
  - 408 Request Timeout
  - 429 Too Many Requests
  - 500 Internal Server Error
  - 502 Bad Gateway
  - 503 Service Unavailable
  - 504 Gateway Timeout

- If you base your retry logic on exceptions, the following typically indicate a transient failure where no connection could be established:
  - WebExceptionStatus.ConnectionClosed
  - WebExceptionStatus.ConnectFailure
  - WebExceptionStatus.Timeout
  - WebExceptionStatus.RequestCanceled
- In the case of a service unavailable status, the service might indicate the appropriate delay before retrying in the **Retry-After** response header or a different custom header. Services might also send additional information as custom headers, or embedded in the content of the response.
- Do not retry for status codes representing client errors (errors in the 4xx range) except for a 408 Request Timeout and 429 Too Many Requests.
- Thoroughly test your retry strategies and mechanisms under a range of conditions, such as different network states and varying system loadings.

Retry strategies

The following are the typical types of retry strategy intervals:

- **Exponential**. A retry policy that performs a specified number of retries, using a randomized exponential back off approach to determine the interval between retries. For example:

```
C#
```

```
var random = new Random();

var delta = (int)((Math.Pow(2.0, currentRetryCount) - 1.0) *
             random.Next((int)(this.deltaBackoff.TotalMilliseconds * 0.8),
             (int)(this.deltaBackoff.TotalMilliseconds * 1.2)));
var interval = (int)Math.Min(checked(this.minBackoff.TotalMilliseconds + delta),
                this.maxBackoff.TotalMilliseconds);
retryInterval = TimeSpan.FromMilliseconds(interval);
```

- **Incremental**. A retry strategy with a specified number of retry attempts and an incremental time interval between retries. For example:

```
C#
```

```
retryInterval = TimeSpan.FromMilliseconds(this.initialInterval.TotalMilliseconds +
                (this.increment.TotalMilliseconds * currentRetryCount));
```

- **LinearRetry**. A retry policy that performs a specified number of retries, using a specified fixed time interval between retries. For example:

```
C#
```

```
retryInterval = this.deltaBackoff;
```

Transient fault handling with Polly

Polly is a library to programmatically handle retries and circuit breaker strategies. The Polly project is a member of the .NET Foundation. For services where the client does not natively support retries, Polly is a valid alternative and avoids the need to write custom retry code, which can be hard to implement correctly. Polly also provides a way to trace errors when they occur, so that you can log retries.

More information

- Connection resiliency
- Data Points - EF Core 1.1