# Azure Core shared client library for .NET

Azure.Core provides shared primitives, abstractions, and helpers for modern .NET Azure SDK client libraries. These libraries follow the [Azure SDK Design Guidelines for .NET](https://azure.github.io/azure-sdk/dotnet_introduction.html) and can be easily identified by package and namespaces names starting with 'Azure', e.g. Azure.Storage.Blobs. A more complete list of client libraries using Azure.Core can be found [here](https://github.com/Azure/azure-sdk-for-net#core-services).

Azure.Core allows client libraries to expose common functionality in a consistent fashion, so that once you learn how to use these APIs in one client library, you will know how to use them in other client libraries.

[Source code](https://github.com/Azure/azure-sdk-for-net/tree/main/sdk/core/Azure.Core/src) | [Package (NuGet)](https://www.nuget.org/packages/Azure.Core/) | [API reference documentation](https://docs.microsoft.com/dotnet/api/azure.core)

## Getting started

Typically, you will not need to install Azure.Core; it will be installed for you when you install one of the client libraries using it. In case you want to install it explicitly (to implement your own client library, for example), you can find the NuGet package [here](https://www.nuget.org/packages/Azure.Core).

## Key concepts

The main shared concepts of Azure.Core (and so Azure SDK libraries using Azure.Core) include:

* Configuring service clients, e.g. configuring retries, logging (ClientOptions).
* Accessing HTTP response details (Response, Response<T>).
* Calling long-running operations (Operation<T>).
* Paging and asynchronous streams (AsyncPageable<T>).
* Exceptions for reporting errors from service requests in a consistent fashion. (RequestFailedException).
* Customizing requests (RequestContext).
* Abstractions for representing Azure SDK credentials. (TokenCredentials).

Below, you will find sections explaining these shared concepts in more detail.

### Thread safety

We guarantee that all client instance methods are thread-safe and independent of each other ([guideline](https://azure.github.io/azure-sdk/dotnet_introduction.html#dotnet-service-methods-thread-safety)). This ensures that the recommendation of reusing client instances is always safe, even across threads.

### Additional concepts

[Client options](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/README.md#configuring-service-clients-using-clientoptions) | [Accessing the response](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/README.md#accessing-http-response-details-using-responset) | [Long-running operations](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/README.md#consuming-long-running-operations-using-operationt) | [Handling failures](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/README.md#reporting-errors-requestfailedexception) | [Diagnostics](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Diagnostics.md) | [Mocking](https://learn.microsoft.com/dotnet/azure/sdk/unit-testing-mocking) | [Client lifetime](https://devblogs.microsoft.com/azure-sdk/lifetime-management-and-thread-safety-guarantees-of-azure-sdk-net-clients/)

## Examples

**NOTE:** Samples in this file apply only to packages that follow [Azure SDK Design Guidelines](https://azure.github.io/azure-sdk/dotnet_introduction.html). Names of such packages usually start with Azure.

### Configuring Service Clients Using ClientOptions

Azure SDK client libraries typically expose one or more *service client* types that are the main starting points for calling corresponding Azure services. You can easily find these client types as their names end with the word *Client*. For example, BlockBlobClient can be used to call blob storage service, and KeyClient can be used to access Key Vault service cryptographic keys.

These client types can be instantiated by calling a simple constructor, or its overload that takes various configuration options. These options are passed as a parameter that extends ClientOptions class exposed by Azure.Core. Various service specific options are usually added to its subclasses, but a set of SDK-wide options are available directly on ClientOptions.

SecretClientOptions options = new SecretClientOptions()

{

Retry =

{

Delay = TimeSpan.FromSeconds(2),

MaxRetries = 10,

Mode = RetryMode.Fixed

},

Diagnostics =

{

IsLoggingContentEnabled = true,

ApplicationId = "myApplicationId"

}

};

SecretClient client = new SecretClient(new Uri("http://example.com"), new DefaultAzureCredential(), options);

More on client configuration in [client configuration samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Configuration.md).

### Accessing HTTP Response Details Using Response<T>

*Service clients* have methods that can be used to call Azure services. We refer to these client methods *service methods*. *Service methods* return a shared Azure.Core type Response<T> (in rare cases its non-generic sibling, a raw Response). This type provides access to both the deserialized result of the service call, and to the details of the HTTP response returned from the server.

// create a client

var client = new SecretClient(new Uri("http://example.com"), new DefaultAzureCredential());

// call a service method, which returns Response<T>

Response<KeyVaultSecret> response = await client.GetSecretAsync("SecretName");

// Response<T> has two main accessors.

// Value property for accessing the deserialized result of the call

KeyVaultSecret secret = response.Value;

// .. and GetRawResponse method for accessing all the details of the HTTP response

Response http = response.GetRawResponse();

// for example, you can access HTTP status

int status = http.Status;

// or the headers

foreach (HttpHeader header in http.Headers)

{

Console.WriteLine($"{header.Name} {header.Value}");

}

More on response types in [response samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Response.md).

### Setting up console logging

To create an Azure SDK log listener that outputs messages to console use AzureEventSourceListener.CreateConsoleLogger method.

// Setup a listener to monitor logged events.

using AzureEventSourceListener listener = AzureEventSourceListener.CreateConsoleLogger();

More on logging in [diagnostics samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Diagnostics.md).

### Reporting Errors RequestFailedException

When a service call fails Azure.RequestFailedException would get thrown. The exception type provides a Status property with an HTTP status code and an ErrorCode property with a service-specific error code.

try

{

KeyVaultSecret secret = client.GetSecret("NonexistentSecret");

}

// handle exception with status code 404

catch (RequestFailedException e) when (e.Status == 404)

{

// handle not found error

Console.WriteLine("ErrorCode " + e.ErrorCode);

}

More on handling responses in [response samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Response.md).

### Consuming Service Methods Returning AsyncPageable<T>

If a service call returns multiple values in pages, it would return Pageable<T>/AsyncPageable<T> as a result. You can iterate over AsyncPageable directly or in pages.

// call a service method, which returns AsyncPageable<T>

AsyncPageable<SecretProperties> allSecretProperties = client.GetPropertiesOfSecretsAsync();

await foreach (SecretProperties secretProperties in allSecretProperties)

{

Console.WriteLine(secretProperties.Name);

}

For more information on paged responses, see [Pagination with the Azure SDK for .NET](https://docs.microsoft.com/dotnet/azure/sdk/pagination).

### Consuming Long-Running Operations Using Operation<T>

Some operations take long time to complete and require polling for their status. Methods starting long-running operations return \*Operation<T> types.

The WaitForCompletionAsync method is an easy way to wait for operation completion and get the resulting value.

// create a client

SecretClient client = new SecretClient(new Uri("http://example.com"), new DefaultAzureCredential());

// Start the operation

DeleteSecretOperation operation = await client.StartDeleteSecretAsync("SecretName");

Response<DeletedSecret> response = await operation.WaitForCompletionAsync();

DeletedSecret value = response.Value;

Console.WriteLine(value.Name);

Console.WriteLine(value.ScheduledPurgeDate);

More on long-running operations in [long-running operation samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/LongRunningOperations.md).

### Customizing Requests Using RequestContext

Besides general configuration of *service clients* through ClientOptions, it is possible to customize the requests sent by *service clients* using protocol methods or convenience APIs that expose RequestContext as a parameter.

var context = new RequestContext();

context.AddClassifier(404, isError: false);

Response response = await client.GetPetAsync("pet1", context);

More on request customization in [RequestContext samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/RequestContext.md).

### Mocking

One of the most important cross-cutting features of our new client libraries using Azure.Core is that they are designed for mocking. Mocking is enabled by:

* providing a protected parameterless constructor on client types.
* making service methods virtual.
* providing APIs for constructing model types returned from virtual service methods. To find these factory methods look for types with the *ModelFactory* suffix, e.g. SecretModelFactory.

For example, the ConfigurationClient.Get method can be mocked (with [Moq](https://github.com/moq/moq4)) as follows:

// Create a mock response

var mockResponse = new Mock<Response>();

// Create a mock value

var mockValue = SecretModelFactory.KeyVaultSecret(

SecretModelFactory.SecretProperties(new Uri("http://example.com"))

);

// Create a client mock

var mock = new Mock<SecretClient>();

// Setup client method

mock.Setup(c => c.GetSecret("Name", null, default))

.Returns(Response.FromValue(mockValue, mockResponse.Object));

// Use the client mock

SecretClient client = mock.Object;

KeyVaultSecret secret = client.GetSecret("Name");

More on mocking in [Unit testing and mocking with the Azure SDK for .NET](https://learn.microsoft.com/dotnet/azure/sdk/unit-testing-mocking).

## Distributed tracing with OpenTelemetry

Azure SDKs are instrumented for distributed tracing using [OpenTelemetry](https://opentelemetry.io/). Distributed tracing allows to follow request through multiple services, record how long network or logical call take along with structured properties describing such operations.

More on diagnostics in [diagnostics samples](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Diagnostics.md).

To setup distributed tracing for your application follow your observability vendor documentation. If you use Azure Monitor, follow the [Start Monitoring Application](https://learn.microsoft.com/azure/azure-monitor/app/opentelemetry-enable?tabs=aspnetcore) guide.

## Troubleshooting

Three main ways of troubleshooting failures are [inspecting exceptions](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Response.md#handling-exceptions), enabling [logging](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Diagnostics.md#Logging), and [distributed tracing](https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/core/Azure.Core/samples/Diagnostics.md#Distributed-tracing)

## Next steps

Explore and install [available Azure SDK libraries](https://azure.github.io/azure-sdk/releases/latest/dotnet.html).

## Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit <https://cla.microsoft.com>.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repositories using our CLA.

This project has adopted the [Microsoft Open Source Code of Conduct](https://opensource.microsoft.com/codeofconduct). For more information see the [Code of Conduct FAQ](https://opensource.microsoft.com/codeofconduct/faq/) or contact [opencode@microsoft.com](mailto:opencode@microsoft.com) with any additional questions or comments.