# Contributing

| Component | Build Status |

| -------------------- | --------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------- |

| Management Libraries | [![Build Status](https://dev.azure.com/azure-sdk/public/\_apis/build/status/net/net%20-%20mgmt%20-%20ci?branchName=master)](https://dev.azure.com/azure-sdk/public/\_build/latest?definitionId=529&branchName=master) |

# Prerequisites:

- Install VS 2019 (Community or higher) and make sure you have the latest updates (https://www.visualstudio.com/).

- Need at least .NET Framework 4.6.1 and 4.7 development tools

- Install the \*\*.NET Core cross-platform development\*\* workloads in VisualStudio

- Install \*\*.NET Core 3.1.101 SDK\*\* or higher for your specific platform. (https://dotnet.microsoft.com/download/dotnet-core/3.1)

- Install the latest version of git (https://git-scm.com/downloads)

- Install [NodeJS](https://nodejs.org/en/) (13.x.x) if you plan to use [C# code generation](https://github.com/Azure/autorest.csharp).

## GENERAL THINGS TO KNOW:

\*\*Client Libraries\*\* are sdks used to interact with azure resources at application run time while \*\*Management Libraries\*\* are those used to manage (create/modify/delete) Azure resources.

\*\*Build Repo :\*\* To build all Client and Management libraries together. Invoke `dotnet build build.proj` from the root of the repo.<br/>To scope to a single service supply a scope property e.g. `dotnet build build.proj /p:Scope=servicebus`. This will build both client and management projects in the specified service. If using msbuild you must run restore first. See below for how to build the [client](#client-libraries) or [management](#management-libraries) libraries independently.

\*\*Path Length :\*\* To account for the \*\*260 characters Path Length Limit\*\* encountered on windows OS, file paths within the repo is keep below 210 characters. This gives you a runway of 49 characters as clone path for your repo. Paths longer that 260 characters will cause the build to break on windows OS and on CI. Assuming you clone to the default VisualStudio location such that the root of your clone is `C:\Users\\\*\*USERNAME\*\*\Source\Repos\azure-sdk-for-net` your username will have to be 9 characters long to avoid build errors caused by long paths. Consider using `C:\git` as you clone path.

\*\*Dependencies :\*\* To ensure that the same versions of dependencies are used for all projects in the repo, package versions are managed from a central location in `eng\Packages.Data.props`. When adding package references you should first ensure that an \*\*Update\*\* reference of the package with the version exist in the \*\*Packages.Data.props\*\* then \*\*Include\*\* the reference without the version in your .csproj. Contact [azuresdkengsysteam@microsoft.com](mailto:azuresdkengsysteam@microsoft.com) if you need to change versions for packages already present in \*\*Packages.Data.props\*\*

\*\*Line Endings :\*\* If working on windows OS ensure git is installed with `Checkout Windows-style, commit Unix-style` option or `core.autocrlf` set to \*true\* in git config. If working on Unix based Linux or MacOS ensure git is installed with `Checkout as-is, commit Unix-style` option or `core.autocrlf` set to \*input\* in git config

# Management Libraries

## TO BUILD:

1. Open any solution, eg "SDKs\Compute\Microsoft.Azure.Management.Compute.sln"

2. Build solution from Visual Studio

### Single Service from Command Line

1. Open Developer Command Prompt

2. From the root directory

3. Invoke `msbuild eng\mgmt.proj /p:scope=Compute`

> \_Build\_ without any scope will build all management SDK's.

### Create single nuget package

In order to build one package and run it's test

`msbuild eng\mgmt.proj /t:CreateNugetPackage /p:scope=Compute`

Nuget package will be created in root directory under \artifacts\packages\Debug (default configuration is Debug)

## TO TEST:

### Using Visual Studio:

1. Build project in Visual Studio.

2. \*\*Test Explorer\*\* window will get populated with tests. Select test and `Run` or `Debug`

### Using the command line:

Run e.g. `msbuild eng\mgmt.proj /t:"Runtests" /p:Scope=Compute`

In the above example \_RunTests\_ will build and run tests for Compute only or you can use command line CLI

`dotnet test Compute\Microsoft.Azure.Management.Compute\tests\Microsoft.Azure.Management.Tests.csproj`

### Non-Windows command line build

Now you can use the same command on non-windows as above for e.g. on Ubuntu you can do something like below:

- `dotnet msbuild eng\mgmt.proj /t:Build /p:scope=Compute`

- `dotnet msbuild eng\mgmt.proj /t:RunTests /p:scope=Compute`

- `dotnet msbuild eng\mgmt.proj /t:CreateNugetPackage /p:scope=Compute`

- `dotnet msbuild build.proj /t:Util /p:UtilityName=InstallPsModules`

### Update build tools

Build tools are now downloaded as part of a nuget package under `root\restoredPackages\microsoft.internal.netsdkbuild.mgmt.tools`

If for any reason there is an update to the build tools, you will then need to first delete directory `root\restoredPackages\microsoft.internal.netsdkbuild.mgmt.tools` and re-execute your build command. This will simply get the latest version of build tools.

## TO CREATE NEW LIBRARY USING TEMPLATE

We have created a dotnet template to make creating new management SDK library easier than ever.

See (README file)[(https://github.com/Azure/azure-sdk-for-net/blob/master/eng/templates/README.md)].

# Client Libraries

## TO BUILD:

### Single Service from Command Line

1. Open Developer Command Prompt

2. Navigate to service directory e.g. \_"sdk\eventhub"\_

3. Invoke `dotnet build`

4. or Build the \*\*service.proj\*\* in the repo root, passing the directory name of the specific service as a property. e.g. `dotnet build eng\service.proj /p:ServiceDirectory=eventhub`

### Single Service from Visual Studio

1. Open any data-plane solution e.g. \_"sdk\eventhub\Microsoft.Azure.EventHubs.sln"\_

2. Build solution from Visual Studio

### All Client Services from Command Line

1. Open Developer Command Prompt

2. Navigate to repository root directory

3. Invoke `dotnet build eng\service.proj`

### Support for Visual Studio Code & Dev Containers

This repository has been configured with support for Visual Studio Code's dev container extension to make it easier to get started working on code without needing to know about how to setup all the pre-requisites. Configuration for dev containers is contained within the ```.devcontainer``` folder off the root directory.

To get started:

1. Install and configure Docker for your platform.

2. Install the [Remote Development extension](https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack) into Visual Studio Code.

3. Clone the repository to your local workstation.

4. Open Visual Studio Code at the root of the reposiory.

5. Select "Reopen in Container" when prompted.

After a few moments of initial configuration Visual Studio Code will launch the container with all dependencies (.NET SDK etc) pre-installed.

## TO TEST:

### Single Service from Command Line

1. Open Developer Command Prompt

2. Navigate to service directory e.g. \_"sdk\eventhub"\_

3. Invoke `dotnet test --filter TestCategory!=Live` \_(Skips live tests)\_

4. or run test against \*\*service.proj\*\* in the repo root, passing the directory name of the specific service as a property. e.g. `dotnet test eng\service.proj /p:ServiceDirectory=eventhub --filter TestCategory!=Live`

### Single Service from Visual Studio

1. Build.

2. Test Explorer window will get populated with tests. Select test and `Run` or `Debug`

### All Client Services from Command Line

1. Open VS 2019 Command Propmpt

2. Navigate to repository root directory

3. Invoke `dotnet test eng\service.proj --filter TestCategory!=Live`

<br/><br/>

### Live testing

Before running or recording live tests you need to create

[live test resources](https://github.com/Azure/azure-sdk-for-net/blob/master/eng/common/TestResources/README.md).

If recording tests, secrets will be sanitized from saved recordings.

If you will be working on contributions over time, you should consider

persisting these variables.

To run live tests after creating live resources:

1. Open Developer Command Prompt

2. Navigate to service directory e.g. \_"sdk\keyvault"\_

3. Invoke `dotnet test`

Some live tests may have additional steps for setting up live testing resources.

See the CONTRIBUTING.md file for the service you wish to test for additional

information or instructions.

### Testing Against Latest Versions of Client Libraries

In some cases, you might want to test against the latest versions of the client libraries. i.e. version not yet published to nuget. For this scenario you should make use of the `UseProjectReferenceToAzureClients` property which when set to `true` will switch all package references for client libraries present in the build to project references. This result in testing against the current version of the libraries in the repo. e.g. `dotnet test eng\service.proj /p:ServiceDirectory=eventhub --filter TestCategory!=Live /p:UseProjectReferenceToAzureClients=true`

## Public API additions

If you make public API changes or additions, the `eng\scripts\Export-API.ps1` script has to be run to update public API listings. This generates a file in the library's directory similar to the example found in `sdk\template\Azure.Template\api\Azure.Template.netstandard2.0.cs`.

Running the script for a project in `sdk\tables` would look like this:

```

eng\scripts\Export-API.ps1 tables

```

## Updating Sample Snippets

If the specific client library has sample snippets in markdown format, they were most likely created with help of the `eng\scripts\Update-Snippets.ps1` script.

Any changes made to the snippet markdown should be done via updating the corresponding C# snippet code and subsequently running the script.

Running the script for a project, for example in `sdk\keyvault`, would look like this:

```

eng\scripts\Update-Snippets.ps1 keyvault

```

When run, the code regions in the format below (where `<snippetName>` is the name of the snippet):

```c#

#region Snippet:<snippetName>

//some sample code

string snippet = "some snippet code";

// Lines prefixed with the below comment format will be ignored by the snippet updater.

/\*@@\*/ string ignored = "this code will not appear in the snippet markdown";

// Lines prefixed with the below comment format will appear in the snippet markdown, but will remain comments in the C#` code.

// Note: these comments should only be used for non-critical code as it will not be compiled or refactored as the code changes.

//@@ snippet = "value that would never pass a test but looks good in a sample!";

#endregion

```

will be mapped to any markdown file with a corresponding code region in the format below where the snippet names match:

\*\*\`\`\`C# Snippet:\<snippetName>\*\*

\*\*\`\`\`\*\*

See the following example of a [snippet C# file](https://github.com/Azure/azure-sdk-for-net/blob/master/sdk/search/Azure.Search.Documents/tests/Samples/HelloWorld.cs) and a [snippet markdown file](https://github.com/Azure/azure-sdk-for-net/blob/master/sdk/search/Azure.Search.Documents/samples/Sample01a\_HelloWorld.md).

Note that snippet names need to be globally unique under a given service directory.

Snippets also can be integrated into XML doc comments. For example:

```c#

/// <summary>

/// Some Property.

/// </summary>

/// <example>

/// This is an example of using a snippet in XML docs.

/// <code snippet="Snippet:<snippetName>">

/// // some sample code.

/// string snippet = "some snippet code";

/// </code>

public string SomeProperty { get; set; }

```

For general information about samples, see the [Samples Guidelines](https://azure.github.io/azure-sdk/dotnet\_introduction.html#dotnet-samples)

## Updating Source on Build

You can run `eng\scripts\Export-API.ps1` and `eng\scripts\Update-Snippets.ps1` simultaneously as part of the build by setting as true either:

1. The property `UpdateSourceOnBuild`

2. The Environment variable `AZURE\_DEV\_UPDATESOURCESONBUILD=true`

e.g.

```

dotnet build eng\service.proj /p:ServiceDirectory=eventhub /p:UpdateSourceOnBuild=true

```

## API Compatibility Verification

.NET is using the [ApiCompat tool](https://github.com/dotnet/arcade/tree/master/src/Microsoft.DotNet.ApiCompat) to enforce API compatibility between versions. Builds of GA'ed libraries will fail locally and in CI if there are breaking changes.

### How it works

We use a dummy project called [ApiCompat](https://github.com/Azure/azure-sdk-for-net/tree/master/eng/ApiCompat/ApiCompat.csproj) to enforce API compatibility between the GA'ed libraries and the most recent version available on Nuget. This project includes package references to the GA'ed library and to Microsoft.DotNet.ApiCompat.

Each library needs to provide a `ApiCompatVersion` property which is set to the last GA'ed version of the library which is used to compare APIs with the current to ensure no breaks

have been introduced.

The `ApiCompatVerification` target defined in `ApiCompat.csproj` is referenced in the [eng/Directory.Build.Data.targets](https://github.com/Azure/azure-sdk-for-net/blob/master/eng/Directory.Build.Data.targets) which causes this target to be executed for each csproj that has the `ApiCompatVersion` property set. For libraries that wish to disable the APICompat check they can remove the `ApiCompatVersion` property from their project. Our version bump automation will automatically add or increment the `ApiCompatVersion` property to the project when it detects that the version it is changing was a GA version which usually indicates that we just shipped that GA version and so it should be the new baseline for api checks.

### Releasing a new version of a GA'ed libary

Since the [eng/Packages.Data.props](https://github.com/Azure/azure-sdk-for-net/blob/master/eng/Packages.Data.props) is currently maintained manually, you will need to update the version number for your library in this file when releasing a new version.

## Dev Feed

We publish daily built packages to a [dev feed](https://dev.azure.com/azure-sdk/public/\_packaging?\_a=feed&feed=azure-sdk-for-net) which can be consumed by adding the dev feed as a package source in Visual Studio.

Follow instructions provided [here](https://docs.microsoft.com/en-us/nuget/consume-packages/install-use-packages-visual-studio#package-sources) and use `https://pkgs.dev.azure.com/azure-sdk/public/\_packaging/azure-sdk-for-net/nuget/v3/index.json` as the source.

You can also achieve this from the command line.

```nuget.exe sources add -Name "Azure SDK for Net Dev Feed" -Source "https://pkgs.dev.azure.com/azure-sdk/public/\_packaging/azure-sdk-for-net/nuget/v3/index.json"```

You can then consume packages from this package source, remember to check the [Include prerelease](https://docs.microsoft.com/en-us/nuget/create-packages/prerelease-packages#installing-and-updating-pre-release-packages) box in Visual Studio when searching for the packages.

# On-boarding New Libraries

### Project Structure

In `sdk\< Service Name >`, you will find projects for services that have already been implemented

1. Client library projects needs to use the $(RequiredTargetFrameworks) \*(defined in eng/Directory.Build.Data.props)\* property in its TargetFramework while management library projects should use $(SdkTargetFx) \_(defined in AzSdk.reference.props)\_

2. Projects of related packages are grouped together in a folder following the structure specified in [Repo Structure](https://github.com/Azure/azure-sdk/blob/master/docs/engineering-system/repo-structure.md)

- Client library packages are in a folder name like \*\*\_Microsoft.Azure.< ServiceName >\_\*\*

- Management library packages are in a folder named like \*\*\_Microsoft.Azure.Management.< Resource Provider Name >\_\*\*

3. Each shipping package contains a project for their \*\*generated\*\* and /or \*\*Customization\*\* code

- The folder \*\*'Generated'\*\* contains the generated code

- The folder \*\*'Customizations'\*\* contains additions to the generated code - this can include additions to the generated partial classes, or additional classes that augment the SDK or call the generated code

- The file \*\*generate.cmd\*\*, used to generate library code for the given package, can also be found in this project

### Standard Process

1. Create fork of [Azure REST API Specs](https://github.com/azure/azure-rest-api-specs)

2. Create fork of [Azure SDK for .NET](https://github.com/azure/azure-sdk-for-net)

3. Create your Swagger specification for your HTTP API. For more information see [Introduction to Swagger - The World's Most Popular Framework for APIs](http://swagger.io)

4. Install the latest version of AutoRest and use it to generate your C# client. For more info on getting started with AutoRest, see the [AutoRest repository](https://github.com/Azure/autorest)

5. Create a branch in your fork of Azure SDK for .NET and add your newly generated code to your project. If you don't have a project in the SDK yet, look at some of the existing projects and build one like the others.

6. \*\*MANDATORY\*\*: Add or update tests for the newly generated code.

7. Once added to the Azure SDK for .NET, build your local package using [client](#client-libraries) or [management](#management-libraries) library instructions shown in the above sections.

8. For management libraries run `eng\scripts\Update-Mgmt-Yml.ps1` to update PR include paths in `eng\pipelines\mgmt.yml`

9. A Pull request of your Azure SDK for .NET changes against \*\*master\*\* branch of the [Azure SDK for .NET](https://github.com/azure/azure-sdk-for-net)

10. The pull requests will be reviewed and merged by the Azure SDK team

### New Resource Provider

1. If you have never created an SDK for your service before, you will need the following things to get your SDK in the repo

2. Follow the standard process described above.

3. Project names helps in using basic heuristics in finding projects as well it's associated test projects during CI process.

4. Create a new directory using the name of your service as specified in [azure-rest-api-specs/specification](https://github.com/Azure/azure-rest-api-specs/tree/master/specification) Repo

5. Follow the the directory structure below

```

sdk\<service name>\<package name>\README.md

sdk\<service name>\<package name>\\*src\*

sdk\<service name>\<package name>\\*tests\*

sdk\<service name>\<package name>\\*samples\*

```

e.g.

```

sdk\eventgrid\Microsoft.Azure.EventGrid\src\Microsoft.Azure.EventGrid.csproj

sdk\eventgrid\Microsoft.Azure.EventGrid\tests\Microsoft.Azure.EventGrid.Tests.csproj

sdk\eventgrid\Microsoft.Azure.Management.EventGrid\src\Microsoft.Azure.Management.EventGrid.csproj

sdk\eventgrid\Microsoft.Azure.Management.EventGrid\tests\Microsoft.Azure.Management.EventGrid.Tests.csproj

```

> Ensure that your service name is the same as it is specified in the [azure-rest-api-specs/specification](https://github.com/Azure/azure-rest-api-specs/tree/master/specification) Repo, that your csproj files starts with \*\*Microsoft.Azure\*\*

> , that test files end with \*\*.Tests\*\* and that management plane project files contain \*\*.Management.\*\*

If you are adding a new service directory, ensure that it is mapped to a friendly name at [ServiceMapping](https://github.com/Azure/azure-sdk-for-net/blob/8c1f53e9099bd5a674f9e77be7e4b1541cd6ab08/doc/ApiDocGeneration/Generate-DocIndex.ps1#L9-L64)

7. Copy .csproj from any other .csproj and update the following information in the new .csproj

| Project Properties |

| ------------------- |

| AssemblyTitle |

| Description |

| VersionPrefix |

| PackageTags |

| PackageReleaseNotes |

| |

> PackageReleaseNotes are important because this information is displayed on www.nuget.org when your nuget package is published

8. Copy existing generate.ps1 file from another service and update the `ResourceProvider` name that is applicable to your SDK. Resource provider refers to the relative path of your REST spec directory in Azure-Rest-Api-Specs repository

During SDK generation, this path helps to locate the REST API spec from the `https://github.com/Azure/azure-rest-api-specs`

# On-boarding New generated code library

1. Make a copy of `/sdk/template/Azure.Template` in you appropriate service directory and rename projects to `Azure.Management.\*` for management libraries or `Azure.\*` (e.g. `sdk/storage/Azure.Management.Storage` or `sdk/storage/Azure.Storage.Blobs`)

2. Modify `autorest.md` to point to you Swagger file or central README.md file. E.g.

``` yaml

input-file:

- https://raw.githubusercontent.com/Azure/azure-rest-api-specs/master/specification/storage/resource-manager/Microsoft.Storage/stable/2019-06-01/blob.json

- https://raw.githubusercontent.com/Azure/azure-rest-api-specs/master/specification/storage/resource-manager/Microsoft.Storage/stable/2019-06-01/file.json

- https://raw.githubusercontent.com/Azure/azure-rest-api-specs/master/specification/storage/resource-manager/Microsoft.Storage/stable/2019-06-01/storage.json

```

``` yaml

require: https://github.com/Azure/azure-rest-api-specs/blob/49fc16354df7211f8392c56884a3437138317d1f/specification/azsadmin/resource-manager/storage/readme.md

```

3. Run `dotnet msbuild /t:GenerateCode` in src directory of the project (e.g. `net\sdk\storage\Azure.Management.Storage\src`). This would run AutoRest and generate the code. (NOTE: this step requires Node 13).

4. For management plan libraries add `azure-arm: true` setting to `autorest.md` client constructors and options would be auto-generated. For data-plane libraries follow the next two steps.

4. Add a `\*ClientOptions` type that inherits from `ClientOptions` and has a service version enum:

``` C#

namespace Azure.Management.Storage

{

public class StorageManagementClientOptions: ClientOptions

{

private const ServiceVersion Latest = ServiceVersion.V2019\_06\_01;

internal static StorageManagementClientOptions Default { get; } = new StorageManagementClientOptions();

public StorageManagementClientOptions(ServiceVersion serviceVersion = Latest)

{

VersionString = serviceVersion switch

{

ServiceVersion.V2019\_06\_01 => "2019-06-01",

\_ => throw new ArgumentOutOfRangeException(nameof(serviceVersion))

};

}

internal string VersionString { get; }

public enum ServiceVersion

{

#pragma warning disable CA1707

V2019\_06\_01 = 1

#pragma warning restore CA1707

}

}

}

```

5. Add public constructors to all the clients using a partial class.

``` C#

public partial class FileSharesClient

{

public FileSharesClient(string subscriptionId, TokenCredential tokenCredential): this(subscriptionId, tokenCredential, StorageManagementClientOptions.Default)

{

}

public FileSharesClient(string subscriptionId, TokenCredential tokenCredential, StorageManagementClientOptions options):

this(new ClientDiagnostics(options), ManagementClientPipeline.Build(options, tokenCredential), subscriptionId, apiVersion: options.VersionString)

{

}

}

```

### Code Review Process

Before a pull request will be considered by the Azure SDK team, the following requirements must be met:

- Prior to issuing the pull request:

- All code must have completed any necessary legal signoff for being publicly viewable (Patent review, JSR review, etc.)

- The changes cannot break any existing functional/unit tests that are part of the central repository.

- This includes all tests, even those not associated with the given feature area.

- Code submitted must have basic unit test coverage, and have all the unit tests pass. Testing is the full responsibility of the service team

- Functional tests are encouraged, and provide teams with a way to mitigate regressions caused by other code contributions.

- Code should be commented.

- Code should be fully code reviewed.

- Code should be able to merge without any conflicts into the dev branch being targeted.

- Code should pass all relevant static checks and coding guidelines set forth by the specific repository.

- All build warnings and code analysis warnings should be fixed prior to submission.

- As part of the pull request (aka, in the text box on GitHub as part of submitting the pull request):

- Proof of completion of the code review and test passes requirements above.

- Identity of QA responsible for feature testing (can be conducted post-merging of the pull request).

- Short description of the payload of pull request.

- After the pull request is submitted:

- Our SLA is 48 hours. When your PR is submitted someone on our team will be auto assigned the PR for review. No need to email us

- MS internal folks, please reach out to us via our Teams channel or

- Send an email to the Azure Developer Platform team [adpteam@microsoft.com](mailto:adpteam@microsoft.com) alias.

- Include all interested parties from your team as well.

- In the message, make sure to acknowledge that the legal signoff process is complete.

Once all of the above steps are met, the following process will be followed:

- A member of the Azure SDK team will review the pull request on GitHub.

- If the pull request meets the repository's requirements, the individual will approve the pull request, merging the code into the dev branch of the source repository.

- The owner will then respond to the email sent as part of the pull request, informing the group of the completion of the request.

- If the request does not meet any of the requirements, the pull request will not be merged, and the necessary fixes for acceptance will be communicated back to the partner team.

### Client Library Tested OSs and .NET Versions

| | Linux (Ubuntu 16.04) | MacOS 10.13 | Windows Server 2016 |

| ----------------- | -------------------- | ----------- | ------------------- |

| \*\*.NET Core 2.1\*\* | x | x | x |

### Issues with Generated Code

Much of the management plane SDK code is generated from metadata specs about the REST APIs. Do not submit PRs that modify generated code. Instead,

- File an issue describing the problem,

- Refer to the the [AutoRest project](https://github.com/azure/autorest) to view and modify the generator, or

- Add additional methods, properties, and overloads to the SDK by adding classes in the 'Customizations' folder of a project

## Versioning

For more information on how we version see [Versioning](doc/dev/Versioning.md)

## Breaking Changes

For information about breaking changes see [Breaking Change Rules](https://github.com/dotnet/corefx/blob/master/Documentation/coding-guidelines/breaking-change-rules.md)