Arcade SDK

Arcade SDK is a set of msbuild props and targets files and packages that provide common build features used across multiple repos, such as CI integration, packaging, VSIX and VS setup authoring, testing, and signing via Microbuild.

The infrastructure of each repository that contributes to .NET Core 3.0 stack is built on top of Arcade SDK. This allows us to orchestrate the build of the entire stack as well as build the stack from source. These repositories are expected to be on the latest version of the Arcade SDK.

Repositories that do not participate in .NET Core 3.0 build may also use Arcade SDK in order to take advantage of the common infrastructure.

The goals are

- to reduce the number of copies of the same or similar props, targets and script files across repos
- enable cross-platform build that relies on a standalone dotnet cli (downloaded during restore) as well as desktop msbuild based build
- be as close to the latest shipping .NET Core SDK as possible, with minimal overrides and tweaks
- be modular and flexible, not all repos need all features; let the repo choose subset of features to import
- unify common operations and structure across repos
- unify Azure DevOps Build Pipelines used to produce official builds

The toolset has four kinds of features and helpers:

- Common conventions applicable to all repos using the toolset.
- Infrastructure required for Azure DevOps CI builds, MicroBuild and build from source.
- Workarounds for bugs in shipping tools (dotnet SDK, VS SDK, msbuild, VS, NuGet client, etc.). Will be removed once the bugs are fixed in the product and the toolset moves to the new version of the tool.
- Abstraction of peculiarities of VSSDK and VS insertion process that are not compatible with dotnet SDK.

The toolset has following requirements on the repository layout.

Single build output

All build outputs are located under a single directory called `artifacts`. The Arcade SDK defines the following output structure:

```
artifacts
  bin
    $(MSBuildProjectName)
      $(Configuration)
  packages
```

```
  $(Configuration)
    Shipping
      $(MSBuildProjectName).$(PackageVersion).nupkg
    NonShipping
      $(MSBuildProjectName).$(PackageVersion).nupkg
    Release
    PreRelease
TestResults
  $(Configuration)
    $(MSBuildProjectName)_$(TargetFramework)_$(TestArchitecture).(xml|html|log|error.log)
VSSetup
  $(Configuration)
    Insertion
      $(VsixPackageId).json
      $(VsixPackageId).vsmand
      $(VsixContainerName).vsix
      $(VisualStudioInsertionComponent).vsman
    DevDivPackages
      $(MSBuildProjectName).$(PackageVersion).nupkg
    $(VsixPackageId).json
    $(VsixContainerName).vsix
VSSetup.obj
  $(Configuration)
    $(VisualStudioInsertionComponent)
SymStore
  $(Configuration)
    $(MSBuildProjectName)
log
  $(Configuration)
    Build.binlog
tmp
  $(Configuration)
obj
  $(MSBuildProjectName)
    $(Configuration)
toolset
```

Having a common output directory structure makes it possible to unify MicroBuild definitions.

directory	description
bin	Build output of each project.
obj	Intermediate directory for each project.
packages	NuGet packages produced by all projects in the repo.

directory	description
VSSetup	Packages produced by VSIX projects in the repo. These packages are experimental and can be used for dogfooding.
VSSetup/Insertion	Willow manifests and VSIXes to be inserted into VS.
VSSetup.obj	Temp files produced by VSIX build.
SymStore	Storage for converted Windows PDBs
log	Build binary log and other logs.
tmp	Temp files generated during build.
toolset	Files generated during toolset restore.

Build scripts and extensibility points

Build scripts

Arcade provides common build scripts in the eng/common folder:

- eng/common/build.ps1|build.sh

  The scripts are designed to be used by repos that need a single `MSBuild` invocation to restore, build, package and test all projects in the repo. These scripts are thin wrappers calling into functions defined in `eng/common/tools.ps1|sh`. If the repository needs to run additional builds or commands it is recommended to create `eng/build.ps1|sh` scripts in the repository using `eng/common/build.ps1|sh` as a template and customize the implementation as necessary. These custom scripts should use common helpers and global variables defined in `eng/common/tools.ps1|sh` and provide command line switches that are a superset of the ones provided by `eng/common/build.ps1|sh`.

- eng/common/tools.ps1|tools.sh

  Defines global variables and functions used in all builds scripts. This includes helpers that install .NET SDK, invoke MSBuild, locate Visual Studio, report build telemetry, etc.

- eng/common/CIBuild.cmd|cibuild.sh

  Repositories that use `eng/common/build.ps1|sh` (as opposed to a customized `eng/build.ps1|sh`) should use this build script for the main build step in their pipeline definition. Repositories

with custom `eng/build.ps1|sh` should also add the corresponding `eng/CIBuild.cmd|cibuild.sh` for use in their pipeline definition.

Since YAML pipeline definition is only executable in CI and not locally on a dev machine the repositories shall minimize the logic implemented in their pipeline definition. Instead of adding more build steps to the pipeline definition that follow `eng/common/CIBuild.cmd` repositories should opt for using the customized build scripts (`eng/build.ps1|sh`) and add the logic there.

Repos may also provide a few convenience build scripts in the repository root that dispatch to either `eng/common/build.ps1|sh` or `eng/build.ps1|sh` (if repo uses customized build scripts) but do not implement any logic:

- Build.cmd | build.sh - default wrapper script for building and restoring the repo.
- Restore.cmd | restore.sh - default wrapper script for restoring the repo.
- Test.cmd | test.sh - default wrapper script for running tests in the repo.

Since the default scripts pass along additional arguments, you could restore, build, and test a repo by running `Build.cmd -test`.

You should feel free to create more repo specific scripts as appropriate to meet common dev scenarios for your repo.

/eng/common/*

The Arcade SDK requires bootstrapper scripts to be present in the repo.

The scripts in this directory shall be present and the same across all repositories using Arcade SDK.

/eng/Build.props

Provide repo specific Build properties such as the list of projects to build.

Arcade project building By default, Arcade builds solutions in the root of the repo. Overriding the default build behavior may be done by either of these methods.

- Provide the project list on the command-line. This will override any list of projects set in `eng/Build.props`.

  Example: `build.cmd -projects MyProject.proj`

  See source code

- Provide a list of projects or solutions in `eng/Build.props`.

  Example:

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
  <ItemGroup>
    <ProjectToBuild Include="$(MSBuildThisFileDirectory)..\MyProject.proj" />
  </ItemGroup>
</Project>
```

Note: listing both project files formats (such as .csproj) and solution files (.sln) at the same time is not currently supported.

Example: specifying a solution to build This is often required for repos which have multiple .sln files in the root directory.

```
<!-- eng/Build.props -->
<Project>
  <ItemGroup>
    <ProjectToBuild Include="$(RepoRoot)MySolution1.sln" />
  </ItemGroup>
</Project>
```

Example: building project files instead of solutions You can also specify a list of projects to build instead of building .sln files.

```
<!-- eng/Build.props -->
<Project>
  <ItemGroup>
    <ProjectToBuild Include="$(RepoRoot)src\**\*.csproj" />
  </ItemGroup>
</Project>
```

Example: conditionally specifying which projects to build You can use custom MSBuild properties to control the list of projects which build.

```
<!-- eng/Build.props -->
<Project>
  <ItemGroup>
    <!-- Usage: build.cmd /p:BuildMyOptionalGroupOfStuff=true -->
    <ProjectToBuild Condition="'$(BuildMyOptionalGroupOfStuff)' == 'true"
                    Include="$(RepoRoot)src\feature1\**\*.csproj" />

    <!-- Only build some projects when building on Windows  -->
    <ProjectToBuild Condition="'$(OS)' == 'Windows_NT"
                    Include="$(RepoRoot)src\**\*.vcxproj" />

    <!-- You can also use MSBuild Include/Exclude syntax -->
    <ProjectToBuild Include="$(RepoRoot)src\**\*.csproj"
                    Exclude="$(RepoRoot)src\samples\**\*.csproj" />
```

```
  </ItemGroup>
</Project>
```

Example: custom implementations of 'Restore' By default, Arcade assumes that the 'Restore' target on projects is implemented using NuGet's restore. If that is not the case, you can opt-out of some Arcade optimizations by setting 'RestoreUsingNuGetTargets' to false.

```
<!-- eng/Build.props -->
<Project>
  <PropertyGroup>
    <RestoreUsingNuGetTargets>false</RestoreUsingNuGetTargets>
  </PropertyGroup>
  <ItemGroup>
    <ProjectToBuild Include="$(RepoRoot)src\dir.proj" />
  </ItemGroup>
</Project>
```

CoreFx does not use the default build projects in its repo - example.

/eng/Versions.props: A single file listing component versions and used tools

The file is present in the repo and defines versions of all dependencies used in the repository, the NuGet feeds they should be restored from and the version of the components produced by the repo build.

```
<Project>
  <PropertyGroup>
    <!-- Base three-part version used for all outputs of the repo (assemblies, packages, vs
    <VersionPrefix>1.0.0</VersionPrefix>
    <!-- Package pre-release label not including build number or the pre-release iteration-
    <PreReleaseVersionLabel>rc</PreReleaseVersionLabel>
    <!-- Package pre-release version iteration. Combines with the label to produce a final
    <PreReleaseVersionIteration>2</PreReleaseVersionIteration>
    <!-- Optional: base short date used for calculating version numbers of release-only pac
    <VersionBaseShortDate>19000</VersionBaseShortDate>

    <!-- Opt-in repo features -->
    <UsingToolVSSDK>true</UsingToolVSSDK>
    <UsingToolIbcOptimization>true</UsingToolIbcOptimization>

    <!-- Opt-out repo features -->
    <UsingToolXliff>false</UsingToolXliff>

    <!-- Versions of other dependencies -->
    <MyPackageVersion>1.2.3-beta</MyPackageVersion>
```

```
  </PropertyGroup>
</Project>
```

The toolset defines a set of tools (or features) that each repo can opt into or opt out. Since different repos have different needs the set of tools that will be imported from the toolset can be controlled by `UsingTool{tool-name}` properties, where *tool-name* is e.g. `Xliff`, `SourceLink`, `XUnit`, `VSSDK`, `IbcOptimization`, etc. These properties shall be set in the Versions.props file.

The toolset also defines default versions for various tools and dependencies, such as MicroBuild, XUnit, VSSDK, etc. These defaults can be overridden in the Versions.props file.

See DefaultVersions for a list of *UsingTool* properties and default versions.

/eng/Tools.props (optional)

Specify package references to additional tools that are needed for the build. These tools are only used for build operations performed outside of the repository solution (such as additional packaging, signing, publishing, etc.).

/eng/Signing.props (optional)

Customization of Authenticode signing process.

Configurable item groups: - `ItemsToSign` List of files to sign in-place, during the build. May list individual files to sign (e.g. .dll, .exe, .ps1, etc.) as well as container files (.nupkg, .vsix, .zip, etc.). All files embedded in a container file are signed (recursively) unless specified otherwise. - `ItemsToSignPostBuild` List of file names (without paths) to sign in post build release pipeline. May only contain files that are published. All files embedded in a container file are signed (recursively) unless specified otherwise. - `FileSignInfo` Specifies Authenticode certificate to use to sign files with given file name. - `FileExtensionSignInfo` Specifies Authenticode certificate to use to sign files with given extension. - `CertificatesSignInfo` Specifies Authenticode certificate properties, such as whether a certificate allows dual signing. - `StrongNameSignInfo` Strong Name key to use to sign a specific managed assembly.

Properties: - `AllowEmptySignList` True to allow ItemsToSign to be empty (the repository doesn't have any file to sign in-build). - `AllowEmptyPostBuildSignList` True to allow ItemsToSignPostBuild to be empty (the repository doesn't have any file to sign post-build). - `PostBuildSign` If true - `ItemsToSignPostBuild` is tracked during the publishing process and these files will be signed during post-build. - `ItemsToSignPostBuild` is populated with the default arcade `ItemsToSign`.

See Signing.md for details.

To change the key used for strong-naming assemblies see `StrongNameKeyId` property.

/eng/Publishing.props (optional)

Customization of publishing process.

/eng/AfterSolutionBuild.targets (optional)

Targets executed in a step right after the solution is built.

/eng/AfterSigning.targets (optional)

Targets executed in a step right after artifacts has been signed.

/global.json

`/global.json` file is present and specifies the version of the dotnet and `Microsoft.DotNet.Arcade.Sdk` SDKs.

For example,

```
{
  "tools": {
    "dotnet": "2.1.400-preview-009088"
  },
  "msbuild-sdks": {
    "Microsoft.DotNet.Arcade.Sdk": "1.0.0-prerelease-63208-02"
  }
}
```

Include `vs` entry under `tools` if the repository should be built via `msbuild` from Visual Studio installation instead of dotnet cli:

```
{
  "tools": {
    "vs": {
      "version": "15.9"
    }
  }
}
```

Optionally, a list of Visual Studio workload component ids may be specified under `vs`:

```
"vs": {
  "version": "15.9",
  "components": ["Microsoft.VisualStudio.Component.VSSDK"]
}
```

If the build runs on a Windows machine that does not have the required Visual Studio version installed the `build.ps1` script attempts to use xcopy-deployable MSBuild package `RoslynTools.MSBuild`. This package will allow the build to

run on desktop msbuild but it may not provide all tools that the repository needs to build all projects and/or run all tests.

The version of `RoslynTools.MSBuild` package can be specified in `global.json` file under `tools` like so:

```
{
  "tools": {
    "vs": {
      "version": "16.0"
    },
    "xcopy-msbuild": "16.0.0-rc1-alpha"
  }
}
```

If it is not specified the build script attempts to find `RoslynTools.MSBuild` version `{VSMajor}.{VSMinor}.0-alpha` where `VSMajor.VSMinor` is the value of `tools.vs.version`.

If the fallback behavior to use xcopy-deployable MSBuild package is not desirable, then a version of `none` should be indicated in `global.json`, like this:

```
{
  "tools": {
    "vs": {
      "version": "16.4"
    },
    "xcopy-msbuild": "none"
  }
}
```

Example: Restoring multiple .NET Core Runtimes for running tests
In /global.json, specify a `runtimes` section and list the shared runtime versions you want installed.

Schema:

```
{
  "tools": {
    "dotnet": "<version>",                                                    // define CLI SDK versi
    "runtimes": {                                                             // optional runtimes se
      "<runtime>": [ "<version>", ..., "<version>" ],
      ...,
      "<runtime>/<architecture>": [ "<version>", ..., "<version>" ]
    }
  }
}
```

`<runtime>` - One of the supported “runtime” values for the dotnet-install script.

`<architecture>` - Optionally include `/<architecture>` when defining the runtime to specify an explicit architecture where "architecture" is one of the supported values for the dotnet-install script. Defaults to "auto" if not specified.

```
{
  "tools": {
    "dotnet": "3.0.100-preview3-010431",
    "runtimes": {
      "dotnet/x64": [ "2.1.7" ],
      "aspnetcore/x64": [ "3.0.0-build-20190219.1" ]
    }
  }
}
```

You may also use any of the properties defined in `eng/Versions.props` to define a version.

Example

```
{
  "tools": {
    "dotnet": "3.0.100-preview3-010431",
    "runtimes": {
      "dotnet/x64": [ "2.1.7", "$(MicrosoftNetCoreAppVersion)" ]
    }
  }
}
```

Note: defining `runtimes` in your global.json will signal to Arcade to install a local version of the SDK for the runtimes to use rather than depending on a matching global SDK.

/NuGet.config

`/NuGet.config` file is present and specifies the MyGet feed to retrieve Arcade SDK from and other feeds required by the repository like so:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <clear />
    <!-- Feed to use to restore the Arcade SDK from -->
    <add key="dotnet-eng" value="https://pkgs.dev.azure.com/dnceng/public/_packaging/dotnet-
    <!-- Feeds to use to restore dependent packages from -->
    <add key="my-feed" value="https://dotnet.myget.org/F/myfeed/api/v3/index.json" />
  </packageSources>
  <disabledPackageSources>
    <clear />
```

```
  </disabledPackageSources>
</configuration>
```

/Directory.Build.props

`Directory.Build.props` shall import Arcade SDK as shown below. The `Sdk.props` file sets various properties and item groups to default values. It is recommended to perform any customizations *after* importing the SDK.

It is a common practice to specify properties applicable to all (most) projects in the repository in `Directory.Build.props`, e.g. public keys for `InternalsVisibleTo` project items.

```
<Project>
  <Import Project="Sdk.props" Sdk="Microsoft.DotNet.Arcade.Sdk" />
  <PropertyGroup>
    <!-- Public keys used by InternalsVisibleTo project items -->
    <MoqPublicKey>0024000004800000940...</MoqPublicKey>

    <!--
      Specify license used for packages produced by the repository.
      Use PackageLicenseExpressionInternal for closed-source licenses.
    -->
    <PackageLicenseExpression>MIT</PackageLicenseExpression>

    <!--
      Specify an id of the key used to generate strong names of assemblies built from this r
    -->
    <StrongNameKeyId>Microsoft</StrongNameKeyId>
  </PropertyGroup>
</Project>
```

/Directory.Build.targets

`Directory.Build.targets` shall import Arcade SDK. It may specify additional targets applicable to all source projects.

```
<Project>
  <Import Project="Sdk.targets" Sdk="Microsoft.DotNet.Arcade.Sdk" />
</Project>
```

/License.txt

The root of the repository shall include a license file named `license.txt`, `license.md` or `license` (any casing is allowed). It is expected that all packages built from the repository have the same license, which is the license declared in the repository root license file.

If the repository uses open source license it shall specify the license name globally using `PackageLicenseExpression` property, e.g. in Directory.Build.props. If the repository uses a closed source license it shall specify the license name using `PackageLicenseExpressionInternal` property. In this case the closed source license file is automatically added to any package build by the repository.

If `PackageLicenseExpression(Internal)` property is set Arcade SDK validates that the content of the license file in the repository root matches the content of the well-known license file that corresponds to the value of the license expression. This validation can be suppressed by setting `SuppressLicenseValidation` to `true` if necessary (not recommended).

See NuGet documentation for details.

Source Projects

Projects are located under `src` directory under root repo, in any subdirectory structure appropriate for the repo.

Projects shall use `Microsoft.NET.Sdk` SDK like so:

```
<Project Sdk="Microsoft.NET.Sdk">
    ...
</Project>
```

Project name conventions

- Unit test project file names shall end with `.UnitTests` or `.Tests`, e.g. `MyProject.UnitTests.csproj` or `MyProject.Tests.csproj`.
- Integration test project file names shall end with `.IntegrationTests`, e.g. `MyProject.IntegrationTests.vbproj`.
- Performance test project file names shall end with `.PerformanceTests`, e.g. `MyProject.PerformanceTests.csproj`.
- If `source.extension.vsixmanifest` is present next to the project file the project is by default considered to be a VSIX producing project.

Other Projects

It might be useful to create other top-level directories containing projects that are not standard C#/VB/F# projects. For example, projects that aggregate outputs of multiple projects into a single NuGet package or Willow component. These projects should also be included in the main solution so that the build driver includes them in build process, but their `Directory.Build.*` may be different from source projects. Hence the different root directory.

Building source packages

Arcade SDK provides targets for building source packages.

Set `IsSourcePackage` to `true` to indicate that the project produces a source package (along with `IsPackable`, `PackageDescription` and other package properties).

If the project does not have an explicitly provided `.nuspec` file (`NuspecFile` property is empty) setting `IsSourcePackage` to `true` will trigger a target that puts sources contained in the project directory to the `contentFiles` directory of the source package produced by the project.

In addition a `build/$(PackageId).targets` file will be auto-generated that links the sources contained in the package to the source server via a Source Link target. If your package already has a `build/$(PackageId).targets` file set `SourcePackageSourceLinkTargetsFileName` property to a different file name (e.g. `SourceLink.targets`) and import the file from `build/$(PackageId).targets`.

If the project is packaged using a custom `.nuspec` file then the source and targets files must be listed in the `.nuspec` file. The path to the generated Source Link targets file will be available within the `.nuspec` file via variable `$SourceLinkTargetsFilePath$`.

Building VSIX packages (optional)

Building Visual Studio components is an opt-in feature of the Arcade SDK. Property `UsingToolVSSDK` needs to be set to `true` in the `Versions.props` file. You need to build using framework MSBuild when building VSIX packages. To build using framework MSBuild, specify the MSBuild engine as VS when building.

Set `VSSDKTargetPlatformRegRootSuffix` property to specify the root suffix of the VS hive to deploy to.

If `source.extension.vsixmanifest` is present next to a project file the project is by default considered to be a VSIX producing project (`IsVsixProject` property is set to true). A package reference to `Microsoft.VSSDK.BuildTools` is automatically added to such project.

Arcade SDK include build target for generating VS Template VSIXes. Adding `VSTemplate` items to project will trigger the target.

`source.extension.vsixmanifest` shall specify `Experimental="true"` attribute in `Installation` section. The experimental flag will be stripped from VSIXes inserted into Visual Studio.

VSIX packages are built to `VSSetup` directory.

Visual Studio Insertion components (optional)

Repository that builds VS insertion components must set `VisualStudioDropName` global property (see Common steps in Azure DevOps pipeline)

To include the output VSIX of a project in Visual Studio insertion, set the `VisualStudioInsertionComponent` property. Multiple VSIXes can specify the same component name, in which case their manifests will be merged into a single insertion unit.

The Visual Studio insertion manifests and VSIXes are generated during Pack task into `VSSetup\Insertion` directory, where they are picked up by MicroBuild Azure DevOps publishing task during official builds.

Arcade SDK also enables building VS Setup Components from .swr files (as opposed to components comprised of one or more VSIXes). Projects that set `VisualStudioInsertionComponent` but do not have `source.extension.vsixmanifest` are considered to be *swix projects* (`IsSwixProject` property is set to true).

Use `SwrProperty` and `SwrFile` items to define a property that will be substituted in .swr files for given value and the set of .swr files, respectively.

For example,

```
<Project Sdk="Microsoft.NET.Sdk">
   <PropertyGroup>
    <TargetFramework>net46</TargetFramework>
    <VisualStudioInsertionComponent>Microsoft.VisualStudio.ProjectSystem.Managed</VisualStud
  </PropertyGroup>
  <ItemGroup>
    <SwrProperty Include="Version=$(VsixVersion)" />
    <SwrProperty Include="VisualStudioXamlRulesDir=$(VisualStudioXamlRulesDir)" />
  </ItemGroup>
  <ItemGroup>
    <SwrFile Include="*.swr" />
  </ItemGroup>
</Project>
```

Where .swr file is:

```
use vs

package name=Microsoft.VisualStudio.ProjectSystem.Managed.CommonFiles
        version=$(Version)

vs.localizedResources
  vs.localizedResource language=en-us
                       title="Microsoft VisualStudio Managed Project System Common Files"
                       description="Microsoft VisualStudio ProjectSystem for C#/VB/F#(Manage

folder "InstallDir:MSBuild\Microsoft\VisualStudio\Managed"
  file source="$(VisualStudioXamlRulesDir)Microsoft.CSharp.DesignTime.targets"
  file source="$(VisualStudioXamlRulesDir)Microsoft.VisualBasic.DesignTime.targets"
```

```
  file source="$(VisualStudioXamlRulesDir)Microsoft.FSharp.DesignTime.targets"
  file source="$(VisualStudioXamlRulesDir)Microsoft.Managed.DesignTime.targets"
```

NOTE: By defining `VisualStudioInsertionComponent` in your project you are implicitly opting-in to having all of the assemblies included in that package marked for `NGEN`. If this is not something you want for a given component you may add `<Ngen>false</Ngen>`.

example:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net472</TargetFramework>
    <VisualStudioInsertionComponent>MyVisualStudioInsertionComponent</VisualStudioInsertionC
    <Ngen>false</Ngen>
  </PropertyGroup>
</Project>
```

Common steps in Azure DevOps pipeline

The steps below assume the following variables to be defined:

- `SignType` - the signing type: "real" (default) or "test"
- `BuildConfiguration` - the build configuration "Debug" or "Release"
- `OperatingSystemName` - "Windows", "Linux", etc.
- `VisualStudioDropName` - VS insertion component drop name (if the repository builds these), usually `Products/$(System.TeamProject)/$(Build.Repository.Name)/$(Build.S`

Signing plugin installation

```
- task: MicroBuildSigningPlugin@3
  displayName: Install Signing Plugin
  inputs:
    signType: real
    esrpSigning: true
  condition: and(succeeded(), ne(variables['SignType'], ''))
```

Restoring internal tools

The following task restores tools that are only available from internal feeds.

```
  - task: NuGetCommand@2
    displayName: Restore internal tools
    inputs:
      command: restore
      feedsToUse: config
      restoreSolution: 'eng\common\internal\Tools.csproj'
      nugetConfigPath: 'NuGet.config'
      restoreDirectory: '$(Build.SourcesDirectory)\.packages'
```

The tools are restored conditionally based on which Arcade SDK features the repository uses (these are specified via `UsingToolXxx` properties).

Official build script

```
- script: eng\common\CIBuild.cmd
          -configuration $(BuildConfiguration)
          /p:OfficialBuildId=$(BUILD.BUILDNUMBER)
          /p:VisualStudioDropName=$(VisualStudioDropName) # required if repository builds VS
          /p:DotNetSignType=$(SignType)
          /p:DotNetSymbolServerTokenMsdl=$(microsoft-symbol-server-pat)
          /p:DotNetSymbolServerTokenSymWeb=$(symweb-symbol-server-pat)
  displayName: Build
```

The Build Pipeline needs to link the following variable group:

- DotNet-Symbol-Server-Pats
  - `microsoft-symbol-server-pat`
  - `symweb-symbol-server-pat`

Publishing test results

```
- task: PublishTestResults@2
  displayName: Publish Test Results
  inputs:
    testRunner: XUnit
    testResultsFiles: 'artifacts/$(BuildConfiguration)/TestResults/*.xml'
    mergeTestResults: true
    testRunTitle: 'Unit Tests'
  condition: always()
```

Publishing logs

```
- task: PublishBuildArtifacts@1
    displayName: Publish Logs
    inputs:
      PathtoPublish: '$(Build.SourcesDirectory)\artifacts\log\$(BuildConfiguration)'
      ArtifactName: '$(OperatingSystemName) $(BuildConfiguration)'
    continueOnError: true
    condition: not(succeeded())
```

Publishing VS insertion artifacts to a drop

This step is required for repositories that build VS insertion components.

```
- task: ms-vseng.MicroBuildTasks.4305a8de-ba66-4d8b-b2d1-0dc4ecbbf5e8.MicroBuildUploadVstsDr
  displayName: Upload VSTS Drop
  inputs:
```

```
    DropName: $(VisualStudioDropName)
    DropFolder: 'artifacts\VSSetup\$(BuildConfiguration)\Insertion'
  condition: succeeded()
```

IBC Optimization Data Embedding

The Arcade SDK includes targets that enable IBC optimization data to be embedded to assemblies produced by the repository.

To enable this functionality set `UsingToolIbcOptimization` to `true` in `/eng/Versions.props`.

Typically, not all projects in a repository need IBC data embedded. Set `ApplyNgenOptimization` to `partial` or `full` in a project to indicate that the assemblies produced by the project should get IBC data embedded.

Use `EnableNgenOptimization` property to control when IBC data embedding is gonna be performed. Unless specified otherwise, `EnableNgenOptimization` is set to `true` if `Configuration` is `Release` and `OfficialBuild` is `true`.

The IBC data embedding is performed by an internal tool `ibcmerge.exe` provided by `Microsoft.Dotnet.IbcMerge` package from an internal Azure DevOps feed. The repository build definition thus must invoke Azure DevOps task that restores internal tools in order for IBC data embedding to work. See Restoring internal tools.

Unless the repository is using an IBC data acquisition mechanism built into the Arcade SDK (such as Visual Studio IBC Data Acquisition) the repository must set the value of `IbcOptimizationDataDir` property and its build must ensure that IBC data are present at the location specified by this property before the `Build` target is executed. It is recommended that such logic is implemented in `/eng/Tools.props` via a target chained after the `Restore` target.

`IbcOptimizationDataDir` specifies the directory that contains optimization data (must end with a directory separator). The optimization data directory is expected to have the following structure:

```
$(IbcOptimizationDataDir)path1\{AssemblyFileName1}\{AssemblyFileName1}
$(IbcOptimizationDataDir)path1\{AssemblyFileName1}\Scenario1.ibc
$(IbcOptimizationDataDir)path1\{AssemblyFileName1}\Scenario2.ibc
$(IbcOptimizationDataDir)path2\{AssemblyFileName2}\{AssemblyFileName2}
$(IbcOptimizationDataDir)path2\{AssemblyFileName2}\Scenario1.ibc
$(IbcOptimizationDataDir)path2\{AssemblyFileName2}\Scenario2.ibc
...
```

The assemblies must be exactly those that were used in the training run that produced the IBC data files. One assembly might be present in multiple copies in different subdirectories. These copies must be identical (an assembly is identified by name only). `ApplyOptimizations` target aggregates all IBC data files

present next to the assembly and all of its copies. Multiple flavors of an assembly with the same names (e.g. assemblies produced by a multi-targeted project) are currently not supported. If necessary in future, it is possible to update the target to group assemblies by MVID instead of file name.

During the build IBC data embedding is performed by `ApplyOptimizations` target, which invokes `ibcmerge.exe` tool. The target runs when `EnableNgenOptimization` is `true` and the project sets `ApplyNgenOptimization` to `partial` or `full`. The target consumes item group `OptimizeAssembly`, whose items are full paths to the assemblies to have IBC data embedded. The assemblies are updated in-place. By default, `OptimizeAssembly` is initialized with the path of the intermediate assembly compiled by the project (this is the file the `CoreCompile` target builds to `obj` directory). The project may update `OptimizeAssembly` item group before `ApplyOptimizations` target is executed if it needs to embed IBC data to other assemblies.

Visual Studio IBC Training

Visual Studio Engineering provides an Azure DevOps Release Pipeline that performs IBC training for assemblies shipped with Visual Studio. Arcade SDK enables repositories to retrieve and embed IBC data this system produces as well as generate inputs for the training.

To enable this functionality set `UsingToolVisualStudioIbcTraining` to `true` in `/eng/Versions.props`.

Visual Studio IBC Data Acquisition

Set `EnableNgenOptimization` property globally or in `/eng/Tools.props` to control when IBC data should be acquired.

The IBC data acquisition is performed by an internal tool `drop.exe` provided by `Drop.App` package from an internal Azure DevOps feed. The repository build definition thus must invoke Azure DevOps task that restores internal tools in order for IBC data embedding to work. See Restoring internal tools.

To retrieve the data the tool needs to authenticate to the VS drop storage. The access token necessary for the authentication is passed via `VisualStudioDropAccessToken` property. If the account the official build of the repository is running on has an access to the VS drop storage, the build definition can pass `/p:VisualStudioDropAccessToken=$(System.AccessToken)` to the `/eng/common/CIBuild.cmd` script.

The IBC data drop produced by a training run is identified by the name of the repository, the branch and the build number the trained binaries came from, and a training run id. An example of IBC data identifier is `OptimizationData/dotnet/roslyn/master-vs-deps/20190210.1/935479/1`, where `dotnet/roslyn` is the repository name, `master-vs-deps` is the branch name, `20190210.1` is the build number and `935479/1` is training

run id. The IBC data acquisition implementation in Arcade SDK requires `RepositoryName` and `VisualStudioIbcSourceBranchName` properties to be set globally or in `/eng/Tools.props` file. These properties specify the repository name and the branch name to use to find the IBC data to acquire. Set `VisualStudioIbcDropId` to the combination of build number and training run id (e.g. `20190210.1/935479/1`) to indicate the exact IBC data drop to acquire. If `VisualStudioIbcDropId` is not specified the most recent IBC data published from the specified repository and branch are acquired.

The IBC data are downloaded to `IbcOptimizationDataDir`, which is initialized to `.tools/IbcData`.

Visual Studio Training Inputs Generation

The training process requires: 1) List of scenarios (tests) to run and binaries to train in each scenario 2) VS Bootstrapper that installs a specific build of Visual Studio and the VS insertion components built by the repository official build 3) Test settings (`Training.runsettings` file)

The scenarios are expected to be listed in file `/eng/config/OptProf.json` in the repository. Arcade SDK reads this file and generates corresponding input files to the training process.

The VS Bootstrapper is built by an Azure DevOps task provided by MicroBuild.

The `Training.runsettings` file is generated by Arcade SDK task `VisualStudio.BuildIbcTrainingSettings`.

The following build definition steps are required for successful generation of all training inputs (in the listed order):

```
variables:
  VisualStudio.MajorVersion: 16                          # specify applicable value
  VisualStudio.ChannelName: 'int.d16.0stg'               # specify applicable value
  VisualStudio.IbcSourceBranchName: 'master-vs-deps'     # specify applicable value
  VisualStudio.DropName: Products/$(System.TeamProject)/$(Build.Repository.Name)/$(Build.S

# ...

- script: eng\cibuild.cmd
  -configuration $(BuildConfiguration)
  # ...
  /p:RepositoryName=$(Build.Repository.Name)
  /p:VisualStudioIbcSourceBranchName=$(VisualStudio.IbcSourceBranchName)
  /p:VisualStudioDropAccessToken=$(System.AccessToken)
  /p:VisualStudioDropName=$(VisualStudio.DropName)      # required by VS insertion compone

# ...
```

```yaml
# Publish OptProf configuration files
- task: ms-vscs-artifact.build-tasks.artifactDropTask-1.artifactDropTask@0
  inputs:
    dropServiceURI: 'https://devdiv.artifacts.visualstudio.com'
    buildNumber: 'ProfilingInputs/DevDiv/$(Build.Repository.Name)/$(Build.SourceBranchName
    sourcePath: '$(Build.SourcesDirectory)\artifacts\OptProf\$(BuildConfiguration)\Data'
    toLowerCase: false
    usePat: false
  displayName: 'OptProf - Publish to Artifact Services - ProfilingInputs'
  condition: succeeded()

# Build VS bootstrapper
# Generates $(Build.StagingDirectory)\MicroBuild\Output\BootstrapperInfo.json
- task: ms-vseng.MicroBuildTasks.0e9d0d4d-71ec-4e4e-ae40-db9896f1ae74.MicroBuildBuildVSBoo
  inputs:
    vsMajorVersion: $(VisualStudio.MajorVersion)
    channelName: $(VisualStudio.ChannelName)
    manifests: $(VisualStudio.SetupManifestList)
    outputFolder: '$(Build.SourcesDirectory)\artifacts\VSSetup\$(BuildConfiguration)\Inser
  displayName: 'OptProf - Build VS bootstrapper'
  condition: succeeded()

# Publish run settings
- task: PowerShell@2
  inputs:
    filePath: eng\common\sdk-task.ps1
    arguments: -configuration $(BuildConfiguration)
               -task VisualStudio.BuildIbcTrainingSettings
               /p:VisualStudioDropName=$(VisualStudio.DropName)
               /p:BootstrapperInfoPath=$(Build.StagingDirectory)\MicroBuild\Output\Bootstr
  displayName: 'OptProf - Build IBC training settings'
  condition: succeeded()

# Publish bootstrapper info
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: $(Build.StagingDirectory)\MicroBuild\Output
    ArtifactName: MicroBuildOutputs
    ArtifactType: Container
  displayName: 'OptProf - Publish Artifact: MicroBuildOutputs'
  condition: succeeded()
```

Testing Locally

To test IBC data embedding and IBC training inputs generation locally:

1. Restore internal tools:

```
nuget.exe restore eng\common\internal\Tools.csproj
```

NuGet.exe may ask for credentials. Note: You need a credential provider for NuGet to be able to successfully authenticate.

2. Run build with the following arguments (choose values of `RepositoryName` and `VisualStudioIbcSourceBranchName` as appropriate):

```
build -configuration Release -restore -ci /p:EnableNgenOptimization=true /p:RepositoryName=c
```

Project Properties Defined by the SDK

`SemanticVersioningV1` (bool)

`true` if `Version` needs to respect SemVer 1.0. Default is `false`, which means format following SemVer 2.0.

`StrongNameKeyId` (string)

The id of the key used to generate assembly strong name for signed assemblies (`SignAssembly` is `true`). By default, `SignAssembly` is set to `true` and `StrongNameKeyId` is set to `MicrosoftShared`. Available values are listed in StrongName.targets.

`AssemblyOriginatorKeyFile`, `PublicKey`, `PublicKeyToken`, `DelaySign`, `PublicSign` properties are set based on the value of `StrongNameKeyId`.

`IsShipping`, `IsShippingAssembly`, `IsShippingPackage`, `IsShippingVsix` (bool)

`IsShipping-` properties are project properties that determine which (if any) assets produced by the project are *shipping*. An asset is considered *shipping* if it is intended to be delivered to customers via an official channel. This channel can be NuGet.org, an official installer, etc. Setting this flag to `true` does not guarantee that the asset will actually ship in the next release of the product. It might be decided after the build is complete that although the artifact is ready for shipping it won't be shipped this release cycle.

By default all assets produced by a project are considered *shipping*. Set `IsShipping` to `false` if none of the assets produced by the project are *shipping*. Test projects (`IsTestProject` is `true`) set `IsShipping` to `false` automatically.

Setting `IsShipping` property is sufficient for most projects. Projects that produce both *shipping* and *non-shipping* assets need a finer grained control. Set `IsShippingAssembly`, `IsShippingPackage` or `IsShippingVsix` to `false` if the assembly, package, or VSIX produced by the project is not *shipping*, respectively.

Build targets shall not directly use `IsShipping`. Instead they shall use `IsShippingAssembly`, `IsShippingPackage` and `IsShippingVsix` depending on the asset they are dealing with.

Examples of usage:

- Set `IsShipping` property to `false` in test/build/automation utility projects.
- Set `IsShipping` property to `false` in projects that produce VSIX packages that are only used only within the repository (e.g. to facilitate integration tests or VS F5) and not expected to be installed by customers.
- Set `IsShippingPackage` property to `false` in projects that package *shipping* assemblies in packages that facilitate transport of assets from one repository to another one, which extracts the assemblies and *ships* them in a *shipping* container.

All assemblies, packages and VSIXes are signed by default, regardless of whether they are *shipping* or not.

By default, Portable and Embedded PDBs produced by *shipping* projects are converted to Windows PDBs and published to Microsoft symbol servers.

By default, all *shipping* libraries are localized.

When `UsingToolNuGetRepack` is true *shipping* packages are repackaged as release/pre-release packages to `artifacts\packages\$(Configuration)\Release` and `artifacts\packages\$(Configuration)\PreRelease` directories, respectively.

`IsVisualStudioBuildPackage` (bool)

Set to `true` in projects that build Visual Studio Build (CoreXT) packages. These packages are non-shipping, but their content is shipping. They are inserted into and referenced from the internal DevDiv `VS` repository.

`PublishWindowsPdb` (bool)

`true` (default) if the PDBs produced by the project should be converted to Windows PDB and published to Microsoft symbol servers. Set to `false` to override the default (uncommon).

`ApplyNgenOptimization` (`partial`, `full` or empty)

Set to `partial` or `full` in a shipping project to require IBC optimization data to be available for the project and embed them into the binary during official build. The value of `partial` indicates partial NGEN, whereas `full` means full NGEN optimization.

`NetCurrent/NetPrevious/NetMinimum/NetFrameworkMinimum`

Properties that define TargetFramework for use by projects so their targeting easily aligns with the current .NET version in development as well as those that are supported. Arcade will update these properties to match the current supported .NET versions, as well as the release being currently developed.

- NetCurrent - The TFM of the major release of .NET that the Arcade SDK aligns with.
- NetPrevious - The previously released version of .NET (e.g. this would be net7 if NetCurrent is net8)
- NetMinimum - Lowest supported version of .NET the time of the release of NetCurrent. E.g. if NetCurrent is net8, then NetMinimum is net6
- NetFrameworkMinimum - Lowest supported version of .NET Framework the time of the release of NetCurrent. E.g. if NetCurrent is net8, then NetFrameworkMinimum is net462

`SkipTests` (bool)

Set to `true` in a test project to skip running tests.

`TestArchitectures` (list of strings)

List of test architectures (`x64`, `x86`) to run tests on. If not specified by the project defaults to the value of `PlatformTarget` property, or `x64` if `Platform` is `AnyCPU` or unspecified.

For example, a project that targets `AnyCPU` can opt-into running tests using both 32-bit and 64-bit test runners on .NET Framework by setting `TestArchitectures` to `x64;x86`.

`TestResultsLogDir` (string)

An alternative path where to save test standard output logs (e.g., `MyProject.Tests_tfm_arch.log`). If not specified, these logs will be saved under the path specified in `$(ArtifactsLogDir)` variable.

`TestTargetFrameworks` (list of strings)

By default, the test runner will run tests for all frameworks a test project targets. Use `TestTargetFrameworks` to reduce the set of frameworks to run against.

For example, consider a project that has `<TargetFrameworks>netcoreapp3.1;net472</TargetFrameworks>`. To only run .NET Core tests run

```
msbuild Project.UnitTests.csproj /t:Test /p:TestTargetFrameworks=netcoreapp3.1
```

To specify multiple target frameworks on command line quote the property value like so:

```
msbuild Project.UnitTests.csproj /t:Test /p:TestTargetFrameworks="netcoreapp3.1;net472"
```

`TestRuntime` (string)

Runtime to use for running tests. Currently supported values are: `Core` (.NET Core), `Full` (.NET Framework) and `Mono` (Mono runtime).

For example, the following runs .NET Framework tests using Mono runtime:

```
msbuild Project.UnitTests.csproj /t:Test /p:TestTargetFrameworks=net472 /p:TestRuntime=Mono
```

`TestRunnerAdditionalArguments` (string)

Additional command line arguments passed to the test runner (e.g. `xunit.console.exe`).

`TestRuntimeAdditionalArguments` (string)

Additional command line arguments passed to the test runtime (i.e. `dotnet` or `mono`). Applicable only when `TestRuntime` is `Core` or `Mono`.

For example, to invoke Mono with debug flags `--debug` (to get stack traces with line number information), set `TestRuntimeAdditionalArguments` to `--debug`. To override the default Shared Framework version that is selected based on the test project TFM, set `TestRuntimeAdditionalArguments` to `--fx-version x.y.z`.

`TestTimeout` (int)

Timeout to apply to an individual invocation of the test runner (e.g. `xunit.console.exe`) for a single configuration. Integer number of milliseconds.

`GenerateResxSource` (bool)

When set to `true`, Arcade will generate a class source for all embedded .resx files.

If source should only be generated for some .resx files, this can be turned on for individual files like this:

```
<ItemGroup>
   <EmbeddedResource Update="MyResources.resx" GenerateSource="true" />
</ItemGroup>
```

The contents of the generated source can be fine-tuned with these additional settings.

`GenerateResxSourceEmitFormatMethods` (bool) When a string in the resx file has argument placeholders, generate a `.FormatXYZ(...)` method with parameters for each placeholder in the string.

Example: if the resx file contains a string “This has {0} and {1} placeholders”, this method will be generated:

```
class Resources
{
  // ...
  public static string FormatMyString(object p0, object p1) { /* ..uses string.Format()... *
}
```

`GenerateResxSourceIncludeDefaultValues` (bool) If set to `true` calls to GetResourceString receive a default resource string value.

`GenerateResxSourceOmitGetResourceString` (bool) If set to `true` the GetResourceString method is not included in the generated class and must be specified in a separate source file.

`FlagNetStandard1XDependencies` (bool) If set to `true` the `FlagNetStandard1xDependencies` target validates that the dependency graph doesn't contain any netstandard1.x packages.

Was this helpful?