C# Code Reviews

Style Guide

Developers should follow Microsoft's C# Coding Conventions and, where applicable, Microsoft's Secure Coding Guidelines.

Code Analysis / Linting

We strongly believe that consistent style increases readability and maintainability of a code base. Hence, we are recommending analyzers / linters to enforce consistency and style rules.

Project Setup

We recommend using a common setup for your solution that you can refer to in all the projects that are part of the solution. Create a `common.props` file that contains the defaults for all of your projects:

```
<Project>
...
    <ItemGroup>
        <PackageReference Include="Microsoft.CodeAnalysis.NetAnalyzers"
Version="5.0.3">
          <PrivateAssets>all</PrivateAssets>
          <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
        </PackageReference>
        <PackageReference Include="StyleCop.Analyzers" Version="1.1.118">
          <PrivateAssets>all</PrivateAssets>
          <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
        </PackageReference>
    </ItemGroup>
    <PropertyGroup>
        <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
    </PropertyGroup>
    <ItemGroup Condition="Exists('$(MSBuildThisFileDirectory)../.editorconfig')"
>
        <AdditionalFiles Include="$(MSBuildThisFileDirectory)../.editorconfig" />
    </ItemGroup>
```

```
...
</Project>
```

You can then reference the `common.props` in your other project files to ensure a consistent setup.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <Import Project="..\common.props" />
</Project>
```

The .editorconfig allows for configuration and overrides of rules. You can have an .editorconfig file at project level to customize rules for different projects (test projects for example).

Details about the configuration of different rules.

.NET analyzers

Microsoft's .NET analyzers has code quality rules and .NET API usage rules implemented as analyzers using the .NET Compiler Platform (Roslyn). This is the replacement for Microsoft's legacy FxCop analyzers.

Enable or install first-party .NET analyzers.

If you are currently using the legacy FxCop analyzers, migrate from FxCop analyzers to .NET analyzers.

StyleCop Analyzer

The StyleCop analyzer is a nuget package (StyleCop.Analyzers) that can be installed in any of your projects. It's mainly around code style rules and makes sure the team is following the same rules without having subjective discussions about braces and spaces. Detailed information can be found here: StyleCop Analyzers for the .NET Compiler Platform.

The minimum rules set teams should adopt is the Managed Recommended Rules rule set.

Automatic Code Formatting

Use .editorconfig to configure code formatting rules in your project.

Build Validation

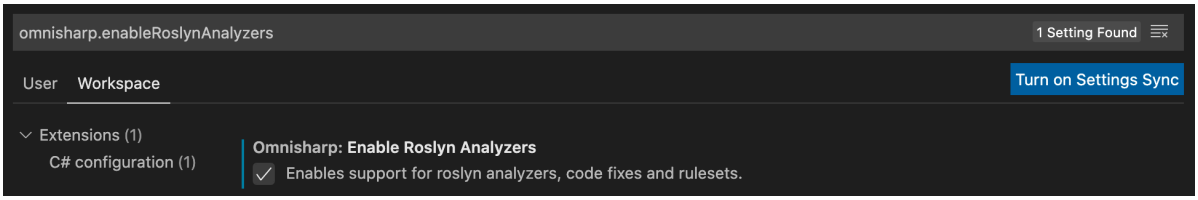
It's important that you enforce your code style and rules in the CI to avoid any team member merging code that does not comply with your standards into your git repo.

If you are using FxCop analyzers and StyleCop analyzer, it's very simple to enable those in the CI. You have to make sure you are setting up the project using nuget and .editorconfig (see Project setup). Once you have this setup, you will have to configure the pipeline to build your code. That's pretty much it. The FxCop analyzers will run and report the result in your build pipeline. If there are rules that are violated, your build will be red.

```
    - task: DotNetCoreCLI@2
      displayName: 'Style Check & Build'
      inputs:
        command: 'build'
        projects: '**/*.csproj'
```

Enable Roslyn Support in VSCode

The above steps also work in VS Code provided you enable Roslyn support for Omnisharp. The setting is `omnisharp.enableRoslynAnalyzers` and must be set to `true`. After enabling this setting you must "Restart Omnisharp" (this can be done from the Command Palette in VS Code or by restarting VS Code).



Code Review Checklist

In addition to the Code Review Checklist you should also look for these C# specific code review items

- [ ] Does this code make correct use of asynchronous programming constructs, including proper use of `await` and `Task.WhenAll` including CancellationTokens?
- [ ] Is the code subject to concurrency issues? Are shared objects properly protected?
- [ ] Is dependency injection (DI) used? Is it setup correctly?
- [ ] Are middleware included in this project configured correctly?

- [ ] Are resources released deterministically using the IDispose pattern? Are all disposable objects properly disposed (using pattern)?
- [ ] Is the code creating a lot of short-lived objects. Could we optimize GC pressure?
- [ ] Is the code written in a way that causes boxing operations to happen?
- [ ] Does the code handle exceptions correctly?
- [ ] Is package management being used (NuGet) instead of committing DLLs?
- [ ] Does this code use LINQ appropriately? Pulling LINQ into a project to replace a single short loop or in ways that do not perform well are usually not appropriate.
- [ ] Does this code properly validate arguments sanity (i.e. CA1062)? Consider leveraging extensions such as Ensure.That
- [ ] Does this code include telemetry (metrics, tracing and logging) instrumentation?
- [ ] Does this code leverage the options design pattern by using classes to provide strongly typed access to groups of related settings?
- [ ] Instead of using raw strings, are constants used in the main class? Or if these strings are used across files/classes, is there a static class for the constants?
- [ ] Are magic numbers explained? There should be no number in the code without at least a comment of why this is here. If the number is repetitive, is there a constant/enum or equivalent?
- [ ] Is proper exception handling set up? Catching the exception base class (`catch (Exception)`) is generally not the right pattern. Instead, catch the specific exceptions that can happen e.g., `IOException` .
- [ ] Is the use of #pragma fair?
- [ ] Are tests arranged correctly with the **Arrange/Act/Assert** pattern and properly documented in this way?
- [ ] If there is an asynchronous method, does the name of the method end with the `Async` suffix?
- [ ] If a method is asynchronous, is `Task.Delay` used instead of `Thread.Sleep` ? `Task.Delay` is not blocking the current thread and creates a task that will complete without blocking the thread, so in a multi-threaded, multi-task environment, this is the one to prefer.
- [ ] Is a cancellation token for asynchronous tasks needed rather than bool patterns?
- [ ] Is a minimum level of logging in place? Are the logging levels used sensible?
- [ ] Are internal vs private vs public classes and methods used the right way?
- [ ] Are auto property set and get used the right way? In a model without constructor and for deserialization, it is ok to have all accessible. For other classes usually a private set or internal

set is better.

- [ ] Is the `using` pattern for streams and other disposable classes used? If not, better to have the `Dispose` method called explicitly.
- [ ] Are the classes that maintain collections in memory, thread safe? When used under concurrency, use lock pattern.

Last update: August 22, 2024