# Contributing to YamlDotNet

\*\*Welcome!\*\*

Thanks for your interest in contributing to this project. Any contribution will

be gladly accepted, provided that they are generally useful and follow the

conventions of the project.

If you are considering a contribution, please read and follow these guidelines.

## Pull requests

All contributions should be submitted as pull requests.

1. Please create \*\*one pull request for each feature\*\*. This results in smaller pull requests that are easier to review and validate.

1. \*\*Avoid reformatting existing code\*\* unless you are making other changes to it.

\* Cleaning-up of `using`s is acceptable, if you made other changes to that file.

\* If you believe that some code is badly formatted and needs fixing, isolate that change in a separate pull request.

1. Always add one or more \*\*unit tests\*\* that prove that the feature / fix you are submitting is working correctly.

1. Please \*\*describe the motivation\*\* behind the pull request. Explain what was the problem / requirement. Unless the implementation is self-explanatory, also describe the solution.

\* Of course, there's no need to be too verbose. Usually one or two lines will be enough.

## Project organization

The main project, YamlDotNet.csproj, is organized in three main namespaces: `Core`, `RepresentationModel` and `Serialization`. The `Core` namespace contains everything that is related to reading and writing YAML. The `RepresentationModel` has classes that represent a YAML stream, similar to XmlDocument for XML. The `Serialization` namespace contains classes to serialize and deserialize object graphs to / from YAML.

Unit tests are all contained in the project named YamlDotNet.Test.csproj.

The PerformanceTests folder contains various projects that contain performance tests that compare various versions of YamlDotNet to detect the impact of new features.

## Building / multiplatform

The project uses a [cake](http://cakebuild.net/) script to specify the build recipe.

If you are on Windows, use the `build.ps1` script to build the project:

```

.\build.ps1 -Target Package

```

You should see an [output similar to this](https://ci.appveyor.com/project/aaubry/yamldotnet/build/4.2.1#L15).

If you are on Linux, use `build.sh`:

```

.\build.sh --target Package

```

Alternatively, if you want to avoid installing the build tools, there is another script that uses a docker container to build. Just replace `build.sh` by `docker-build.sh`:

```

.\docker-build.sh --target Package

```

### Build targets

The following table describes the most important build targets:

| Target | Description | Example |

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

| Clean | Deletes the build output. | `.\build.ps1 -Target Clean` |

| Build | Builds a single configuration. | `.\build.ps1 -Target Build -Configuration Release-Unsigned` |

| Test | Runs unit tests on a single configuration. | `.\build.ps1 -Target Test -Configuration Release-Unsigned` |

| Build-Release-Configurations | Builds all the release configurations. | `.\build.ps1 -Target Build-Release-Configurations` |

| Test-Release-Configurations | Runs unit tests on all the release configurations. | `.\build.ps1 -Target Test-Release-Configurations` |

| Package | Build the NuGet package. | `.\build.ps1 -Target Package` |

| Document | Generates the samples documentation. | `.\build.ps1 -Target Document` |

Building for Unity requires installing

[Visual Studio Tools for Unity](https://visualstudiogallery.msdn.microsoft.com/20b80b8c-659b-45ef-96c1-437828fe7cf2/file/92287/8/Visual%20Studio%202013%20Tools%20for%20Unity.msi).

### Target platforms

The project targets the following platforms:

\* .NET Framework 4.5

\* .NET Framework 3.5

\* .NET Framework 2.0

\* .NET Standard 2.1

\* .NET Standard 1.3

\* Unity Subset v3.5

In the csproj, the `TargetFrameworks` element also targets the following platforms for technical reasons:

\* net40: this is a hack used to target Unity. That target is overriden and in reality it targets Unity Subset v3.5.

\* .NET Core 3.0: this is to benefit from nullable annotations in the BCL.

### Build configurations

The following table describes the available build configurations:

| Configuration | Description |

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

| Debug | Default debug build. |

| Release | Release build. |

| Debug-AOT | Builds the AOT tests project, that tests compatibility with mono's AOT compilation. |

There are a few differences between the various target platforms,

mainly in the reflection API. In order to adapt the code to each platform,

`#if ... #endif` sections are used. When possible, such sections should be placed

in the `Helpers/Portability.cs` file. An effective technique is to define an extension

method that is used through the code, and has different implementations depending

on the build variables.

## AOT compatibility

Some platforms - such as IOS - forbid dynamic code generation. This prevents Just-in-Time compilation (JIT) from being used. In those cases, one can use Mono's Ahead-of-Time compilation (AOT). This results on a precompiled assembly that does not rely on JIT. There are [some limitations](http://www.mono-project.com/docs/advanced/aot/#limitation-generic-interface-instantiation) however, most of them related to usage of generics.

In order to ensure that YamlDotNet is compatible with AOT compilation, an automatic test has been created that runs on every commit on [Travis CI](https://travis-ci.org/aaubry/YamlDotNet). That test exercises the serializer and deserializer to help identify AOT-related problems.

## Coding style

Attempt to follow the [SOLID](https://en.wikipedia.org/wiki/SOLID\_%28object-oriented\_design%29) principles. In particular, try to give each type a single responsibility, and favor composition to combine features.

As long as you keep the code readable, I don't care too much about any specific coding convention. There are only a few rules that should be honored:

\* Use \*\*4 spaces\*\* to indent.

\* Each class / interface / struct / delegate \*\*goes to its own file\*\*.

\* The only acceptable exception is for small and closely related types.

\* Use sane indentation rules. Break long lines when needed, but don't be obsessive:

\* This is \*\*OK\*\*:

```C#

Traverse(

new ObjectDescriptor(

value.Value,

underlyingType,

value.Type,

value.ScalarStyle

),

visitor,

currentDepth

);

```

\* This is \*\*OK too\*\*:

```C#

Traverse(

new ObjectDescriptor(value.Value, underlyingType, value.Type, value.ScalarStyle),

visitor,

currentDepth

);

```

\* This is \*\*not very good\*\*:

```C#

Traverse(new ObjectDescriptor(value.Value, underlyingType, value.Type, value.ScalarStyle), visitor, currentDepth);

```

\* This is \*\*awful\*\*:

```C#

Traverse(new ObjectDescriptor(value.Value,

underlyingType,

value.Type,

value.ScalarStyle),

visitor,

currentDepth);

```