DOCFX PDF SAMPLE

Table of Contents

Getting Started with `docfx`

Getting Started

This is a seed.

`docfx` is an API documentation generator for .NET, currently support C# and VB. It has the ability to extract triple slash comments out from your source code. What's more, it has syntax to link additional files to API to add additional remarks. `docfx` will scan your source code and your additional conceptual files and generate a complete HTML documentation website for you. `docfx` provides the flexibility for you to customize the website through templates. We currently have several embedded templates, including websites containing pure static html pages and also website managed by AngularJS.

- Click "View Source" for an API to route to the source code in GitHub (your API must be pushed to GitHub)
- `docfx` provide DNX version for cross platform use.
- `docfx` can be used within Visual Studio seamlessly. **NOTE** offical `docfx.msbuild` nuget package is now in pre-release version. You can also build your own with source code and use it locally.
- We support **Docfx Flavored Markdown(DFM)** for writing conceptual files. DFM is **100%** compatible with *Github Flavored Markdown(GFM)* and add several new features including *file inclusion*, *cross reference*, and *yaml header*.

Engineering Guidelines

Basics

Copyright header and license notice

All source code files require the following exact header according to its language (please do not make any changes to it).

> extension: **.cs**

```
// Licensed to the .NET Foundation under one or more agreements.
// The .NET Foundation licenses this file to you under the MIT license.
```

> extension: **.js**

```
// Licensed to the .NET Foundation under one or more agreements.
// The .NET Foundation licenses this file to you under the MIT license.
```

> extension: **.css**

```
/**
 * Licensed to the .NET Foundation under one or more agreements.
 * The .NET Foundation licenses this file to you under the MIT license.
 */
```

> extension: **.tmpl**, **.tmpl.partial**

```
{{!Licensed to the .NET Foundation under one or more agreements. The .NET Foundation lic
```

External dependencies

This refers to dependencies on projects (i.e. NuGet packages) outside of the `docfx` repo, and especially outside of Microsoft. Adding new dependencies require additional approval.

Current approved dependencies are:

- Newtonsoft.Json
- Jint
- HtmlAgilityPack

- Nustache
- YamlDotNet

Code reviews and checkins

To help ensure that only the highest quality code makes its way into the project, please submit all your code changes to GitHub as PRs. This includes runtime code changes, unit test updates, and deployment scripts. For example, sending a PR for just an update to a unit test might seem like a waste of time but the unit tests are just as important as the product code and as such, reviewing changes to them is also just as important.

The advantages are numerous: improving code quality, more visibility on changes and their potential impact, avoiding duplication of effort, and creating general awareness of progress being made in various areas.

In general a PR should be signed off(using the 👍 emoticon) by the Owner of that code.

To commit the PR to the repo **do not use the Big Green Button**. Instead, do a typical push that you would use with Git (e.g. local pull, rebase, merge, push).

Source Code Management

Branch strategy

In general:

- `master` has the code for the latest release on NuGet.org. (e.g. `1.0.0`, `1.1.0`)
- `dev` has the code that is being worked on but not yet released. This is the branch into which devs normally submit pull requests and merge changes into. We run daily CI towards `dev` branch and generate pre-release nuget package, e.g. `1.0.1-alpha-9-abcdefsd`.
- `hotfix` has the code for fixing `master` bug after it is released. `hotfix` changes will be merged back to `master` and `dev` once it is verified.

Solution and project folder structure and naming

Solution files go in the repo root. The default entry point is `All.sln`.

Every project also needs a `project.json` and a matching `.xproj` file. This `project.json` is the source of truth for a project's dependencies and configuration options.

Solution need to contain solution folders that match the physical folder (`src`, `test`, `tools`, etc.).

Assembly naming pattern

The general naming pattern is `Docfx.<area>.<subarea>`.

Unit tests

We use *xUnit.net* for all unit testing.

Coding Standards

Please refer to C# Coding standards for detailed guideline for C# coding standards.

TODO Template Coding standards

TODO Template Preprocess JS Coding standards

C# Coding Standards

Introduction

The coding standard will be used in conjunction with customized version of *StyleCop* and *FxCop* [**TODO**] during both development and build process. This will help ensure that the standard is followed by all developers on the team in a consistent manner.

> "Any fool can write code that a computer can understand. Good programmers write code that humans understand".
>
> Martin Fowler. *Refactoring: Improving the design of existing code.*

Purpose

The aim of this section is to define a set of C# coding standards to be used by CAPS build team to guarantee maximum legibility, reliability, re-usability and homogeneity of our code. Each section is marked *Mandatory* or *Recommended*. *Mandatory* sections, will be enforced during code reviews as well as tools like *StyleCop* and *FxCop*, and code will not be considered complete until it is compliant.

Scope

This section contains general C# coding standards which can be applied to any type of application developed in C#, based on *Framework Design Guidelines*.

It does not pretend to be a tutorial on C#. It only includes a set of limitations and recommendations focused on clarifying the development.

Tools

- Resharper is a great 3rd party code cleanup and style tool.
- StyleCop analyzes C# srouce code to enforce a set of style and consistency rules and has been integrated into many 3rd party development tools such as Resharper.
- FxCop is an application that analyzes managed code assemblies (code that targets the .NET Framework common language runtime) and reports information about the assemblies, such as possible design, localization, performance, and security improvements.
- C# Stylizer does many of the style rules automatically

Highlights of Coding Standards

This section is not intended to give a summary of all the coding standards that enabled by our customized StyleCop, but to give a highlight of some rules one will possibly meet in

daily coding life. It also provides some recommended however not mandatory(which means not enabled in StyleCop) coding standards.

File Layout (Recommended)

Only one public class is allowed per file.

The file name is derived from the class name.

```
Class   : Observer
Filename: Observer.cs
```

Class Definition Order (Mandatory)

The class definition contains class members in the following order, from *less* restricted scope (public) to *more* restrictive (private):

- Nested types, e.g. classes, enum, struct, etc.
- Field members, e.g. member variables, const, etc.
- Member functions
  - Constructors
  - Finalizer (Do not use unless absolutely necessary)
  - Methods (Properties, Events, Operations, Overridables, Static)
  - Private nested types

Naming (Mandatory)

- **DO** use PascalCasing for all public member, type, and namespace names consisting of multiple words.

```
PropertyDescriptor
HtmlTag
IOStream
```

NOTE: A special case is made for two-letter acronyms in which both letters are capitalized, e.g. *IOStream*

- **DO** use camelCasing for parameter names.

```
propertyDescriptor
htmlTag
ioStream
```

- **DO** start with underscore for private fields

```
private readonly Guid _userId = Guid.NewGuid();
```

- **DO** start static readonly fields, constants with capitalized case

```
private static readonly IEntityAccessor EntityAccessor = null;
private const string MetadataName = "MetadataName";
```

- **DO NOT** capitalize each word in so-called closed-form compound words.
- **DO** have **"Async"** explicitly in the Async method name to notice people how to use it properly

Formatting (Mandatory)

- **DO** use spaces over tabs, and always show all spaces/tabs in IDE

> **Tips**
>
> Visual Studio > TOOLS > Options > Text Editor > C# > Tabs > Insert spaces (Tab size: 4)
>
> Visual Studio > Edit > Advanced > View White Space

- **DO** add *using* inside *namespace* declaration

```
namespace Microsoft.Content.Build.BuildWorker.UnitTest
{
      using System;
}
```

- **DO** add a space when:

1. `for (var i = 0; i < 1; i++)`
2. `if (a == b)`

Cross-platform coding

Our code should supports multiple operating systems. Don't assume we only run (and develop) on Windows. Code should be sensitvie to the differences between OS's. Here are some specifics to consider.

- **DO** use `Enviroment.NewLine` instead of hard-coding the line break instead of `\r\n`, as Windows uses `\r\n` and OSX/Linux uses `\n`.

> **Note**
>
> Be aware that thes line-endings may cause problems in code when using `@""` text blocks with line breaks.

- **DO** Use `Path.Combine()` or `Path.DirectorySeparatorChar` to separate directories. If this is not possible (such as in scripting), use a forward slash `/`. Windows is more forgiving than Linux in this regard.

Unit tests and functional tests

Assembly naming

The unit tests for the `Microsoft.Foo` assembly live in the `Microsoft.Foo.Tests` assembly.

The functional tests for the `Microsoft.Foo` assmebly live in the `Microsoft.Foo.FunctionalTests` assmebly.

In general there should be exactly one unit test assebmly for each product runtime assembly. In general there should be one functional test assembly per repo. Exceptions can be made for both.

Unit test class naming

Test class names end with `Test` and live in the same namespace as the class being tested. For example, the unit tests for the `Microsoft.Foo.Boo` class would be in a `Microsoft.Foo.Boo` class in the test assembly.

Unit test method naming

Unit test method names must be descriptive about *what is being tested, under what conditions, and what the expectations are*. Pascal casing and underscores can be used to improve readability. The following test names are correct:

```
PublicApiArgumentsShouldHaveNotNullAnnotation
Public_api_arguments_should_have_not_null_annotation
```

The following test names are incorrect:

```
Test1
Constructor
```

```
FormatString
GetData
```

Unit test structure

The contents of every unit test should be split into three distinct stages, optionally separated by these comments:

```
// Arrange
// Act
// Assert
```

The crucial thing here is the `Act` stage is exactly one statement. That one statement is nothing more than a call to the one method that you are trying to test. keeping that one statement as simple as possible is also very important. For example, this is not ideal:

```
int result = myObj.CallSomeMethod(GetComplexParam1(), GetComplexParam2(), GetComplexPar
```

This style is not recomended because way too many things can go wrong in this one statement. All the `GetComplexParamN()` calls can throw for a variety of reasons unrelated to the test itself. It is thus unclear to someone running into a problem why the failure occured.

The ideal pattern is to move the complex parameter building into the `Arrange section:

```
// Arrange
P1 p1 = GetComplexParam1();
P2 p2 = GetComplexParam2();
P3 p3 = GetComplexParam3();

// Act
int result = myObj.CallSomeMethod(p1, p2, p3);

// Assert
Assert.AreEqual(1234, result);
```

Now the only reason the line with `CallSomeMethod()` can fail is if the method itself blew up.

Testing exception messages

In general testing the specific exception message in a unit test is important. This ensures that the exact desired exception is what is being tested rather than a different exception of the same type. In order to verify the exact exception it is important to verify the message.

```
var ex = Assert.Throws<InvalidOperationException>(
    () => fruitBasket.GetBananaById(1234));
Assert.Equal(
    "1234",
    ex.Message);
```

Use xUnit.net's plethora of built-in assertions

xUnit.net includes many kinds of assertions – please use the most appropriate one for your test. This will make the tests a lot more readable and also allow the test runner report the best possible errors (whether it's local or the CI machine). For example, these are bad:

```
Assert.Equal(true, someBool);

Assert.True("abc123" == someString);

Assert.True(list1.Length == list2.Length);

for (int i = 0; i < list1.Length; i++) {
    Assert.True(
        String.Equals
            list1[i],
            list2[i],
            StringComparison.OrdinalIgnoreCase));
}
```

These are good:

```
Assert.True(someBool);

Assert.Equal("abc123", someString);

// built-in collection assertions!
Assert.Equal(list1, list2, StringComparer.OrdinalIgnoreCase);
```

Parallel tests

By default all unit test assemblies should run in parallel mode, which is the default. Unit tests shouldn't depend on any shared state, and so should generally be runnable in parallel. If the tests fail in parallel, the first thing to do is to figure out why; do not just disable parallel tests!

For functional tests it is reasonable to disable parallel tests.

Markdown

Markdown is a lightweight markup language with plain text formatting syntax. Docfx supports CommonMark compliant Markdown parsed through the Markdig parsing engine.

Link to Math Expressions

Block Quotes

> This is a block quote.

Alerts

> **NOTE**
> Information the user should notice even if skimming.

> **TIP**
> Optional information to help a user be more successful.

> **IMPORTANT**
> Essential information required for user success.

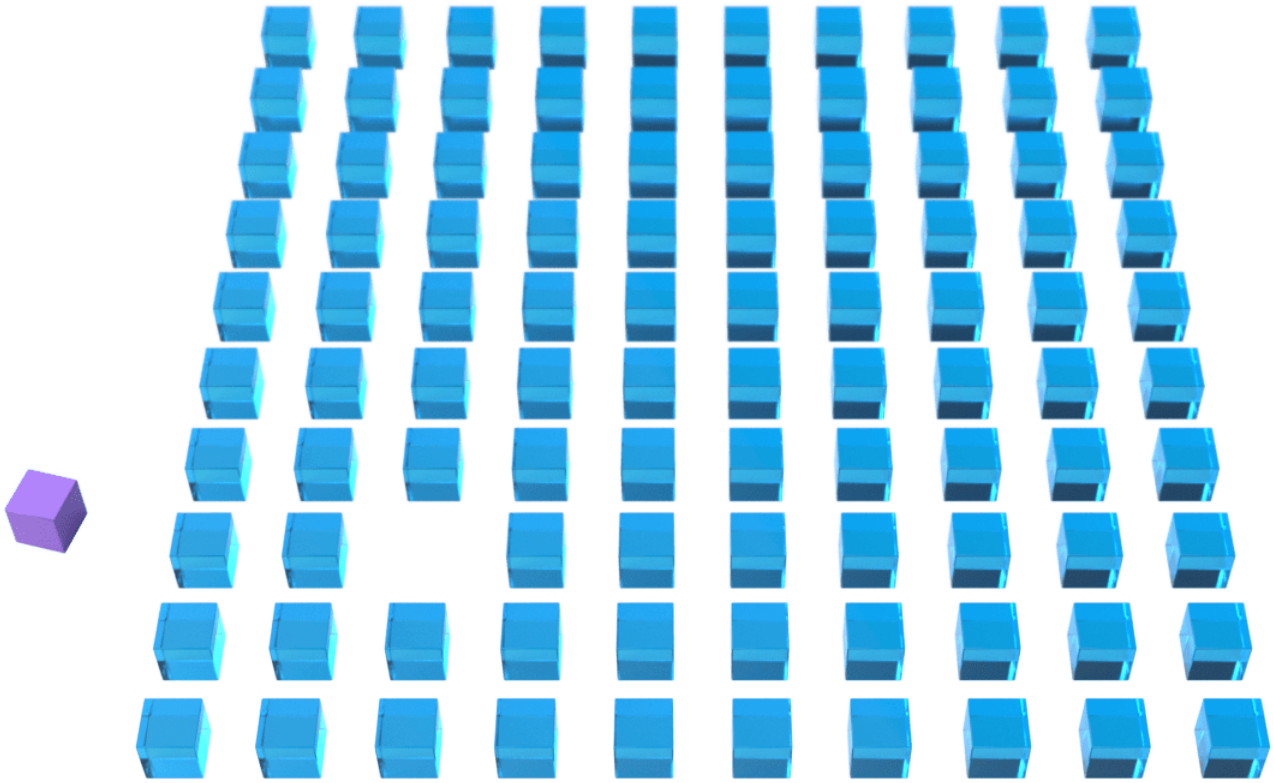
> **CAUTION**
> Negative potential consequences of an action.

> **WARNING**
> Dangerous certain consequences of an action.

MY TODO

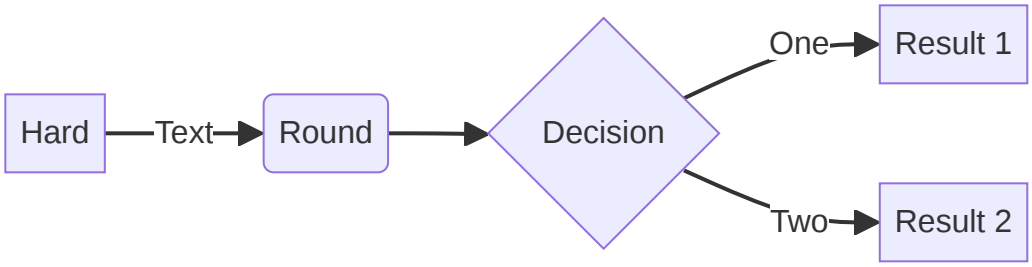
This is a TODO.

Image



Mermaid Diagrams

Flowchart



Code Snippet

The example highlights lines 2, line 5 to 7 and lines 9 to the end of the file.

```
using System;
using Azure;
using Azure.Storage;
using Azure.Storage.Blobs;

class Program
{
    static void Main(string[] args)
    {
        // Define the connection string for the storage account
        string connectionString = "DefaultEndpointsProtocol=https;AccountName=<your-acc

        // Create a new BlobServiceClient using the connection string
        var blobServiceClient = new BlobServiceClient(connectionString);

        // Create a new container
        var container = blobServiceClient.CreateBlobContainer("mycontainer");

        // Upload a file to the container
        using (var fileStream = File.OpenRead("path/to/file.txt"))
        {
            container.UploadBlob("file.txt", fileStream);
        }

        // Download the file from the container
        var downloadedBlob = container.GetBlobClient("file.txt").Download();
        using (var fileStream = File.OpenWrite("path/to/downloaded-file.txt"))
        {
            downloadedBlob.Value.Content.CopyTo(fileStream);
        }
    }
}
```

Math Expressions

This sentence uses `$` delimiters to show math inline: $\sqrt{3x-1}+(1+x)^2$

The Cauchy-Schwarz Inequality

$$\left(\sum_{k=1}^{n} a_k b_k\right)^2 \leq \left(\sum_{k=1}^{n} a_k^2\right) \left(\sum_{k=1}^{n} b_k^2\right)$$

This expression uses `\$` to display a dollar sign: $\sqrt{\$4}$

To split $100 in half, we calculate $100/2$

Custom Syntax Highlighting

```
resource storageAccount 'Microsoft.Storage/storageAccounts@2021-06-01' = {
  name: 'hello'
  // (...)
}
```

Tabs

Linux | Windows

Content for Linux...

The above tab group was created with the following syntax:

```
# [Linux](#tab/linux)

Content for Linux...

# [Windows](#tab/windows)

Content for Windows...

---
```

Tabs are indicated by using a specific link syntax within a Markdown header. The syntax can be described as follows:

```
# [Tab Display Name](#tab/tab-id)
```

A tab starts with a Markdown header, `#`, and is followed by a Markdown link `[]()`. The text of the link will become the text of the tab header, displayed to the customer. In order for the header to be recognized as a tab, the link itself must start with `#tab/` and be followed by an ID representing the content of the tab. The ID is used to sync all same-ID tabs across the page. Using the above example, when a user selects a tab with the link `#tab/windows`, all tabs with the link `#tab/windows` on the page will be selected.

Dependent tabs

It's possible to make the selection in one set of tabs dependent on the selection in another set of tabs. Here's an example of that in action:

.NET | TypeScript | REST API

.NET content for Linux...

Notice how changing the Linux/Windows selection above changes the content in the .NET and TypeScript tabs. This is because the tab group defines two versions for each .NET and TypeScript, where the Windows/Linux selection above determines which version is shown for .NET/TypeScript. Here's the markup that shows how this is done:

```
# [.NET](#tab/dotnet/linux)

.NET content for Linux...

# [.NET](#tab/dotnet/windows)

.NET content for Windows...

# [TypeScript](#tab/typescript/linux)

TypeScript content for Linux...

# [TypeScript](#tab/typescript/windows)

TypeScript content for Windows...

# [REST API](#tab/rest)

REST API content, independent of platform...

---
```